UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior
Ingeniría Técnica de Telecomunicaciones
especialidad en Sonido e Imagen

PROYECTO FIN DE CARRERA

Diseño y desarrollo de una herramienta de simulación en Python para la composición de aplicaciones distribuidas de tiempo real basadas en servicios

Autor
Juan Luis Vélez Valero
Tutora
Iria Manuela Estévez Ayres

Leganés, Febrero 2013

Resumen

En este proyecto fin de carrera se aborda la composición de aplicaciones distribuidas de tiempo real basadas en servicios. Para ello, se han implementado en lenguaje de programación Python dos algoritmos de composición, uno exhaustivo y otro heurístico, para poder evaluar su comportamiento en un sistema distribuido de tiempo real.

Estos algoritmos se han probado para aplicaciones sencillas, con pocas funcionalidades en serie, así como aplicaciones más complejas en las que determinadas funcionalidades tengan que esperar a otras para poder ejecutarse. Las aplicaciones, para ser compuestas, han de pasar una serie de condiciones y ser planificables en el sistema.

Para comprobar la planificabilidad en el sistema se puede usar un algoritmo de planificación exacto (con tiempo de convergencia no acotado) o uno que calcule cotas al tiempo de respuesta (que da condiciones suficientes pero no necesarias en la planificabilidad). En concreto se ha elegido evaluar el comportamiento del algoritmo heurístico con ambos algoritmos, así como la mejora que supone la introducción de un control de acceso basado en cotas al tiempo de respuesta.

En cuanto al lenguaje de programación, se elige Python ya que es un lenguaje orientado a objetos y que soporta diferentes plataformas, además de ser de fácil desarrollo y de tipado dinámico.

Abstract

This final project dissertation deals with the composition of real-time distributed applications based on services. On this purpose, two composition algorithms have been introduced in Python programming language, one exhaustive and the other one heuristic. It has been done in order to be able to evaluate its behaviour in a system distributed in real time.

These algorithms have been tried in simple applications, with few functionalities in series, as well as in more complex applications in which these functionalities need to wait for other ones to be carried out. The applications shall go through a series of different conditions and be predictable in order to be compound.

Either an exact schedulability algorithm (with not bounded convergence time) or another able to calculate the response time level (what gives us enough conditions but not necessary for the predictability) could be use to verify the system predictability. To be more speci fic, it has been decided to evaluate the heuristic algorithm's behaviour with both schedulability algorithms, as well as the improvement what means the introduction of an access control system based on the bounded response-times.

Regarding the programming language, Python has been selected not only because it supports multiple programming paradigms, including object oriented programming, and it is supported on different platforms but also because it uses dynamic typing, and application development is easy and fast.

Nomenclatura

CASH	Capacity Sharing
CBS	Constant Bandwidth Server
DDS	(Deadline Driven Scheduling Algorithm) Algoritmo EDF
DMS	(Deadline Montonic Scheduling) Planificación de Plazo Monótono. Prioridad a la tarea más urgente
EC	(Elementary Cycle) Ciclo elemental
ECU	(Electronic Control Unit) Unidad de control eléctronica
EDF	(Earliest Deadline First) Prioridad a la tarea con el plazo más próximo
FTT	(Flexible Time-Triggered paradigm) Paradigma flexible gobernado por tiempo
GRUB	Greedy Reclamation of Unused Bandwidth
HOPA	Heuristic Optimized Priority Assignment
LSW	(Length Sync Window) Longitud de la ventana de sincronización
PDA	(Processor Demand Analysis) Analísis de demanda de procesador
QoS	(Quality of Service) Calidad de Servicio
RMS	(Rate Montonic Scheduling) Planificación con prioridad a la tarea más frecuente
RTA	(Real Time Analisys) Análisis en tiempo real
RUB	(Response time Upper Bound) Cota superior del tiempo de respuesta
SOA	(Service Oriented Architecture) Arquitectura orientada a servicio
TBS	Total Bandwidth Server

TDA	(Time Dilation Algorithm) Algoritmo de Dilatación Temporal
TM	(Trigger Message) Ene l paradigma FTT, mensaje de disparo
W3C	World Wide Web Consortium
WCDO	Worst Case Analysis for Dynamic Osets
WCDOPS	Worst Case Analysis for Dynamic Osets with Priority Schemes
WCET	(Worst Case Execution Time) Tiempo de ejecución en el peor caso
WCRT	(Worst Case Response Time) Tiempo de Respuesta en el peor caso

Índice general

1. Planteamiento y objetivos **13**

1.1. Introducción . . . 13

1.2. Motivación del proyecto . . . 14

1.3. Objetivos principales . . . 15

1.4. Estructura de la memoria . . . 15

2. Estado del arte **17**

2.1. Sistemas de tiempo real . . . 17

2.1.1. Tareas de tiempo real . . . 18

2.1.2. Planificación de sistemas de tiempo real . . . 19

2.1.3. Planificación centralizada . . . 20

2.1.3.1. Planificación estática . . . 21

2.1.3.2. Planificación dinámica . . . 22

2.1.3.3. Plazos superiores a los periodos . . . 29

2.2. Sistemas distribuidos . . . 29

2.2.1. Planificación distribuida . . . 31

2.2.1.1. Modelo y política de planificación para el análisis de sistemas distribuidos 32

2.2.1.2. Análisis de planificabilidad . . . 33

2.3. Paradigma de servicios . . . 34

2.3.1. Servicios Web . . . 35

2.4. Algoritmo de generación de tareas . . . 36

3. Modelos teóricos y decisiones de diseño **37**

3.1. Modelo del sistema . . . 37

3.2. Modelo de Aplicaciones . . . 38

3.3. Modelo de servicio . . . 42

3.3.1. Implementaciones de servicio . . . 43

3.4. Decisiones de diseño: resumen y conclusiones . . . 48
3.4.1. Modelo del sistema . . . 48
3.4.2. Modelo de aplicaciones . . . 48
3.4.3. Modelo de servicios e implementaciones de servicio . . . 49

4. Algoritmos de composición 50
4.1. Algoritmos usados . . . 50
4.1.1. Algoritmo exhaustivo . . . 51
4.1.2. Algoritmo heurístico . . . 53
4.1.2.1. Heurísticos implementados . . . 55
4.2. Decisiones de diseño de los algoritmos . . . 56
4.2.1. Decisiones previas . . . 57
4.2.2. Decisiones de diseño del algoritmo exhaustivo . . . 58
4.2.3. Decisiones de diseño del algoritmo heurístico . . . 58

5. Programa desarrollado 60
5.1. Introducción . . . 60
5.2. Script principal . . . 60
5.3. Clases . . . 63
5.3.1. Implementacion_Servicio . . . 63
5.3.2. Mensaje . . . 64
5.3.3. Tarea . . . 66
5.4. Funciones . . . 67
5.4.1. Funciones relativas a la estructuración de las implementaciones . . . 67
5.4.2. Funciones relativas a la exploración y obtención de combinaciones . . . 72
5.4.3. Funciones relativas al cumplimiento de los requisitos de la aplicación . . . 77
5.4.4. Funciones relativas a la obtención de resultados . . . 84
5.5. Conclusiones . . . 85

6. Evaluación y resultados 86
6.1. Introducción . . . 86
6.1.1. Definición de las pruebas . . . 86
6.2. Resultados obtenidos . . . 88
6.2.1. Prueba 1 . . . 88
6.2.1.1. Aplicación sencilla. Tres servicios en serie . . . 88
6.2.1.2. Aplicación sencilla. Cinco servicios en serie . . . 92
6.2.1.3. Aplicación compleja. Cinco servicios con paralelos . . . 96
6.2.1.4. Conclusiones . . . 100

6.2.2. Prueba 2 100
6.2.2.1. Aplicación sencilla. Tres servicios en serie 101
6.2.2.2. Aplicación sencilla. Cinco servicios en serie 104
6.2.2.3. Aplicación compleja. Cinco servicios con paralelos 107
6.2.2.4. Conclusiones 110
6.2.3. Prueba 3 110
6.2.3.1. Aplicación sencilla. Tres servicios en serie 111
6.2.3.2. Aplicación sencilla. Cinco servicios en serie 111
6.2.3.3. Aplicación compleja. Cinco servicios con paralelos 112
6.2.3.4. Conclusiones 112
6.2.4. Caso de estudio 113

7. Conclusiones y lineas futuras **116**
7.1. Conclusiones generales 116
7.2. Trabajos Futuros 117
7.2.1. Futuras lineas de investigación 117

Índice de figuras

2.1. Ejemplo de transacción de tiempo real de [9] 32
2.2. Esquema de una arquitectura orientada a servicios de [9] 35

3.1. Ejemplo de aplicación distribuida 40
3.2. Ejemplo de una aplicación distribuida tras ser reconfigurada 41
3.3. Tarea productora 44
3.4. Tarea consumidora 45
3.5. Tarea productora-consumidora 45
3.6. Aplicación con dos puntos de sincronización 47
3.7. Aplicación sencilla 47

5.1. Diagrama de flujo general del programa desarrollado 62
5.2. Clase Implementación_Servicio 64
5.3. Clase Mensaje 65
5.4. Clase Tarea 66
5.5. Diagrama de flujo de la función utilización 79

6.1. Aplicación sencilla con tres servicios en serie 88
6.2. Aplicación sencilla con cinco servicios en serie 92
6.3. Aplicación compleja de cinco servicios 96
6.4. Falsos negativos relativos a una aplicación sencilla de tres servicios 101
6.5. Tiempo de ejecución. App. sencilla de tres servicios y utilización previa al 0 % 102
6.6. Tiempo de ejecución. App. sencilla de tres servicios y utilización previa al 30 % 103
6.7. Tiempo de ejecución. App. sencilla de tres servicios y utilización previa al 55 % 103
6.8. Falsos negativos relativos a una aplicación sencilla de cinco servicios 105
6.9. Tiempo de ejecución. App. sencilla de cinco servicios y utilización previa al 0 % 105
6.10. Tiempo de ejecución. App. sencilla de cinco servicios y utilización previa al 30 % 106
6.11. Tiempo de ejecución. App. sencilla de cinco servicios y utilización previa al 55 % 106

6.12. Falsos negativos relativos a una aplicación compleja con cinco servicios 107
6.13. Tiempo de ejecución. App. compleja de cinco servicios y utilización previa al 0 % 108
6.14. Tiempo de ejecución. App. compleja de cinco servicios y utilización previa al 30 % 109
6.15. Tiempo de ejecución. App. compleja de cinco servicios y utilización previa al 55 % 109
6.16. Tabla de datos del artículo de Zeng y Di Natale 113
6.17. Aplicación propuesta en el artículo de Zeng y Di Natale 114
6.18. Versión simplificada del sistema propuesto por Zeng y Di Natale 114
6.19. Versión esquemática y simplificada de la aplicación propuesta por Zeng y Di Natale . . . 115

Indice de tablas

2.1. Clasificación algoritmos de planificación y ejemplos para planificación centralizada 21

6.1. Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 4 implementaciones por servicio y carga previa 0 % 89
6.2. Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 4 implementaciones por servicio y carga previa 30 % 89
6.3. Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 4 implementaciones por servicio y carga previa 55 % 89
6.4. Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 6 implementaciones por servicio y carga previa 0 % 90
6.5. Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 6 implementaciones por servicio y carga previa 30 % 90
6.6. Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 6 implementaciones por servicio y carga previa 55 % 91
6.7. Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 8 implementaciones por servicio y carga previa 0 % 91
6.8. Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 8 implementaciones por servicio y carga previa 30 % 91
6.9. Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 8 implementaciones por servicio y carga previa 55 % 92
6.10. Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 4 implementaciones por servicio y carga previa 0 % 92
6.11. Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 4 implementaciones por servicio y carga previa 30 % 93
6.12. Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 4 implementaciones por servicio y carga previa 55 % 93
6.13. Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 6 implementaciones por servicio y carga previa 0 % 94

6.14. Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 6 implementaciones por servicio y carga previa 30 % 94
6.15. Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 6 implementaciones por servicio y carga previa 55 % 94
6.16. Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 8 implementaciones por servicio y carga previa 0 % 95
6.17. Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 8 implementaciones por servicio y carga previa 30 % 95
6.18. Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 8 implementaciones por servicio y carga previa 55 % 96
6.19. Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 4 implementaciones por servicio y carga previa 0 % 96
6.20. Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 4 implementaciones por servicio y carga previa 30 % 97
6.21. Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 4 implementaciones por servicio y carga previa 55 % 97
6.22. Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 6 implementaciones por servicio y carga previa 0 % 97
6.23. Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 6 implementaciones por servicio y carga previa 30 % 98
6.24. Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 6 implementaciones por servicio y carga previa 55 % 98
6.25. Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 8 implementaciones por servicio y carga previa 0 % 99
6.26. Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 8 implementaciones por servicio y carga previa 30 % 99
6.27. Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 8 implementaciones por servicio y carga previa 55 % 99
6.28. Datos relativos a los tiempos de respuesta y error introducido de una aplicación sencilla con tres servicios en serie 102
6.29. Datos relativos a los tiempos de respuesta y error introducido de una aplicación sencilla de cinco servicios en serie 104
6.30. Datos relativos a los tiempos de respuesta y error introducido de una aplicación compleja con cinco servicios 108
6.31. Ctrl. de admisión con utilización de los nodos al 55 % para 3 servicios en serie 111
6.32. Ctrl. de admisión con utilización de los nodos al 55 % para cinco servicios en serie 112
6.33. Ctrl. de admisión con utilización de los nodos al 55 % para cinco servicios con paralelos . 112

6.34. Datos relativos a las tareas resultantes para su uso como datos reales. 113
6.35. Resultados relativos al experimento propuesto a partir del artículo de Zeng y Di Natale. . 114

Capítulo 1

Planteamiento y objetivos

Este proyecto fin de carrera tiene como fin la implementación y evaluación de un conjunto de técnicas para la composición de aplicaciones en tiempo real basadas en servicios. Este capítulo describe el contexto en el que se encuentra este proyecto, así como los objetivos planteados para su realización.

1.1. Introducción

Para poder situar el contexto en el que se desarrolla este proyecto hay que definir los sistemas con los que se va a trabajar a lo largo de la elaboración de éste. Un sistema de tiempo real es aquél que no sólo se basa en la correcta lógica y ejecución de las operaciones computacionales de las aplicaciones, sino que también depende de obtener los resultados de dicha ejecución acotados en el tiempo [54, 15]. Si, por cualquier motivo, las restricciones de tiempo no se cumplen, se considera que el sistema ha fallado [9]. Estas restricciones de tiempo son, por tanto, un requisito indispensable a cumplir en un sistema de tiempo real.

Un sistema de tiempo real es predecible, ya que el tiempo en el que un proceso debe finalizar su ejecución para que el sistema no sufra daño alguno asegura esta característica. Además, este tiempo se denomina *plazo* (deadline).

Este documento se centra en sistemas de tiempo real basados en servicios, que aplican conceptos del paradigma de orientación a servicios (SOA, Service Oriented Architecture) en sistemas de tiempo real para dotarlos de flexibilidad y dinamismo. Un servicio, según *Steves Jones*, es "*un dominio de control de una envergadura acotada que contiene un conjunto de tareas que cooperan para alcanzar objetivos relacionados*" [35]. Es decir, a lo largo de este proyecto fin de carrera un servicio es *una entidad software autocontenida que proporciona una determinada funcionalidad.*

Tradicionalmente, las técnicas usadas en sistemas de tiempo real eran bastante pesimistas, no siendo adecuadas para el correcto tratamiento de sistemas donde la flexibilidad y dinamismo son un requisito.

Con las técnicas clásicas se obtenían, con un gasto de recursos importante, tiempos deterministas basados en el tiempo de ejecución y asignación de recursos en el peor caso. Por tanto, es necesario que permita dotar de dinamismo y flexibilidad a sistemas de tiempo real.

Así, la comunidad de tiempo real ha aumentado el número de investigaciones en la adaptación a entornos dinámicos [17], tanto a bajo como a alto nivel, para sustituir el desarrollo tradicional en este entorno.

Con respecto al hardware, los dispositivos que acompañan la vida de las personas tienen más funcionalidades que para las que estaban diseñados inicialmente [23]. Esto conlleva una integración de varias funcionalidades en el mismo aparato, lo que implica que ya no solo hay que satisfacer las necesidades funcionales del usuario, sino que hay que tener en cuenta la situación personal de dicho usuario, es decir, sus intereses, aficiones o circunstancias laborales, entre otros factores [9]. Esto hace que un entorno distribuido evolucione a un entorno omnipresente, lo que implica, de nuevo, el paradigma de computación orientada a servicios mencionado anteriormente.

1.2. Motivación del proyecto

La mayoría de los sistemas de tiempo real actuales trabajan con planificación estática, es decir, seleccionan previamente un conjunto de servicios fijos, que son asignados a los recursos disponibles.

En un sistema de tiempo real basado en servicios no sólo es importante el desarrollo de algoritmos de composición de aplicaciones, sino que dichos algoritmos deben estar acotados en tiempo, es decir, la consecución de la aplicación requerida debe cumplir unas condiciones temporales, que limitan al sistema, para considerarse de tiempo real. Es decir, un sistema que, frente a un soporte tradicional de planificación estática, proporcione soporte a la actualización dinámica y reconfiguración de servicios, como por ejemplo:

- **Actualización dinámica.** Uso de de las nuevas versiones de los servicios que aparecen para mejorar una aplicación dada, en el caso de que mejore el rendimiento.
- **Tolerancia a fallos.** Las distintas implementaciones de servicio son usadas como implementaciones de seguridad, preparadas para ser usadas si uno de los nodos físicos se ve afectado por un fallo y hubiera que componer una nueva aplicación.
- **Equilibrio de carga.** El sistema intenta conseguir una carga de los nodos físicos equitativa para evitar una sobrecarga de alguno de ellos y evitar una disminución de la calidad de las aplicaciones.

Los algoritmos de composición diseñados por *Estévez Ayres* [9], aunque estaban acotados en número de combinaciones, podían no estar acotados temporalmente, ya que, al usar tiempos de respuesta exactos dependían fuertemente de la convergencia del análisis de planificabilidad.

El uso de tests para el cálculo del tiempo de respuesta aproximado en lugar de exacto, genera un tiempo lineal que no depende de la carga del sistema ni de otros parámetros, como el orden de magnitud del rango

de períodos [26]. Sin embargo, estos tests tienen como contrapartida que ofrecen una cota superior del tiempo de respuesta, lo que puede indicar que un determinado conjunto de tareas no es planificable, cuando en realidad sí que lo es.

Por lo tanto, la inclusión y análisis de este tipo de tests, que pueden ser concebidos como un control de admisión ligero en los algoritmos de composición ya existentes, son de interés pues de esta forma se limita no sólo el número de combinaciones exploradas, sino el tiempo de respuesta del algoritmo de composición, haciendo viable aplicarlo cuando el sistema está en ejecución.

1.3. Objetivos principales

Basándose en los estudios de Iria Estévez Ayres, [9], los principales objetivos fijados para la realización de este proyecto son:

- Estudio y análisis de los algoritmos usados en sistemas de tiempo real distribuidos, así como los empleados por *Estévez Ayres* en su tesis.
- Implementación de dichos algoritmos en lenguaje de programación Python, ya que es un lenguaje de programación que soporta orientación a objetos, es multiplataforma, de tipado dinámico y de fácil desarrollo.
- Validación del comportamiento de los algoritmos propuestos:
  - Simulación de los algoritmos implementados para las distintas configuraciones.
  - Obtención y validación de las cotas temporales teóricas.
  - Estudio de qué algoritmo emplear dependiendo de la estructura de la aplicación.
- Implementación de una herramienta que facilita la interacción con los algoritmos implementados a lo largo del proyecto, así como la realización de las pruebas necesarias para la presentación de resultados.
- Implementación de una mejora de los algoritmos de planificación (RUB), así como un estudio y análisis de cómo influye esta mejora en los falsos negativos y en el error respecto a haber usado tests de planificación clásicos (RTA) y respecto al algoritmo de búsqueda exhaustivo.

1.4. Estructura de la memoria

La presente memoria está organizada por capítulos, los cuales se desglosan de la siguiente manera:

- Capítulo 2. Se presentan los tipos de sistemas, técnicas y métodos relacionados con este trabajo.

- Capítulo 3. En este capítulo se trata el modelo de sistema y el modelo de aplicación de tiempo real basada en servicios. También se presenta el concepto de implementación de servicio, proponiendo un ejemplo de aplicación para la elección de unas implementaciones determinadas. Además, se incluyen las decisiones de diseño tomadas para cada modelo.

- Capítulo 4. Se introducen los algoritmos de composición inicial de aplicaciones. Primero se presenta un algoritmo exhaustivo que realiza la búsqueda de una combinación inicial óptima según una figura de mérito, solventando la problemática de la composición de aplicaciones en tiempo real. Dada la complejidad computacional de un algoritmo de estas características, hace inviable su uso en sistemas de tiempo real, por lo que se propone un algoritmo mejorado que, en base a acotación de las combinaciones a explorar, ofrece una solución subóptima.

- Capítulo 5. Este capítulo recoge todas las clases y funciones implementadas en Python para la realización del presente proyecto. Además, las funciones están organizadas por bloques, según su cometido.

- Capítulo 6. En este capítulo se proponen tres pruebas para evaluar las prestaciones de los algoritmos implementados. Además, se incluye un caso particular de un caso real basado en un artículo.

- Capítulo 7. Se presentan las conclusiones generales extraídas a lo largo de la elaboración de este proyecto, así como una sugerencia de posibles lineas de trabajo.

Capítulo 2

Estado del arte

En este capítulo se presentan los conceptos teóricos sobre las tecnologías más importantes relacionadas con este trabajo. Primero se presentarán las tecnologías sobre sistemas de tiempo real centralizados, y posteriormente los sistemas distribuidos. Se estudiará con especial interés los algoritmos de planificación de las tareas en ambas configuraciones, al ser la parte fundamental en la que se basa el proyecto. Asimismo se introducirán las redes a utilizar a la hora de trabajar con sistemas distribuidos. Finalmente se presentará el paradigma de la orientación a servicios.

2.1. Sistemas de tiempo real

Un sistema de tiempo real es aquél en el que el instante en que se produce la salida del mismo es representativo. Esto quiere decir que las operaciones se ejecutan dentro de un intervalo de tiempo específico, convirtiéndose en un sistema en el que el funcionamiento no sólo depende de que las acciones sean correctas [16, 54]. El tiempo en el que el proceso puede ejecutarse sin causar anomalías en el sistema se conoce como plazo.

Los procesos se pueden clasificar como firmes, flexibles o críticos, atendiendo a la importancia del resultado de agotar el plazo [37, 51]. Los primeros son los procesos que una vez agotado el plazo, ya no son útiles; los flexibles son los que, a pesar terminar de ejecutarse fuera de plazo, siguen teniendo un papel importante en el sistema; y por último, los procesos críticos, son aquellos que en caso de no ejecutarse dentro de su plazo puede causar un error fatal en el sistema.

Los sistemas de tiempo real también se clasifican según su comportamiento con respecto al plazo. Se conoce como sistemas de tiempo real duro (*hard real-time*) a los sistemas que necesitan finalizar todos sus procesos antes terminar su plazo, y como sistemas de tiempo real suave (*soft real-time*) a los que permiten la finalización de las operaciones fuera de su plazo.

Si un sistema de tiempo real procesa al menos un proceso crítico, se convierte en un sistema de tiempo

real crítico, mientras que en cualquier otro caso, se conoce como sistema de tiempo real flexible. Un ejemplo de un sistema crítico es un láser en una operación de ojos, y un ejemplo de un sistema flexible es un traspaso de información digital entre dos ordenadores.

Además, en estos sistemas se intenta obtener una utilización óptima y se deben cumplir unos requisitos de tiempo.

2.1.1. Tareas de tiempo real

Hay que tener en cuenta que en la naturaleza y en el mundo real las acciones pueden ocurrir a la vez y no solo linealmente, es decir, ocurren en paralelo. Esto desemboca en que para diseñar un sistema que intenta asemejarse a la realidad, éste tiene que consistir en una serie de tareas que interacciones entre ellas y con el entorno [4, 12].

Los procesos (o tareas, indistintamente), pueden clasificarse como *periódicos* o *aperiódicos*, dependiendo de sus características temporales. Son periódicos cuando se ejecutan con cierta periodicidad fija. Los aperiódicos, a su vez, pueden ser esporádicos, y se caracterizan por la tasa de llegada al sistema.

Todos los procesos, a pesar de ser ejecutados por el mismo *software*, pueden tener restricciones temporales diferentes, ser independientes entre si, o requerir acceso a distintos recursos compartidos y no solo al procesador.

Por ello, los procesos tienen una serie de características que han de conocerse para un correcto funcionamiento del sistema que se implementa en un momento dado. Un conjunto de N tareas periódicas Π se puede caracterizar como:

$$\Pi = \{\tau_i = (C_i, D_i, T_i, \phi_i, p_i), i = 1 \dots N\} \tag{2.1}$$

donde,

- C_i es el tiempo ejecución en el peor caso de la tarea τ_i; también es conocido como WCET (Worst Case Execution Time).
- D_i es el plazo relativo de la tarea τ_i. Una vez que la tarea es invocada, debe finalizar, como muy tarde, en D_i unidades de tiempo, con $D_i \leq T_i$.
- T_i es el periodo de la tarea τ_i. Es el tiempo entre dos invocaciones sucesivas de la tarea τ_i. Cada tarea puede producir una secuencia infinita de invocaciones separadas T_i unidades de tiempo.
- ϕ_i es el desplazamiento de la tarea τ_i, con $\phi_i \leq T_i$. Es el desplazamiento respecto a $t = 0$ del momento de invocación de la tarea. La primera invocación de la tarea se realiza en $t = \phi_i$, y las siguientes estarán separadas T_i unidades de tiempo.
- p_i es la prioridad de la tarea τ_i, siendo p_i la mas alta y p_N la mas baja. Tradicionalmente se usaba como prioridad el subíndice de la tarea, pero con los sistemas actuales no tiene sentido, ya que una

tarea puede tener dos prioridades distintas, o varias tareas tener la misma prioridad. De ahí surgió el la necesidad de introducir la prioridad como parámetro.

En el caso particular de que los procesos sean esporádicos, se puede seguir empleando esta notación, cambiando el periodo por el periodo mínimo de activación, y dejando de tener en cuenta el desplazamiento temporal:

$$\sigma_i = (C_i, D_i, T_{min_i}, p_i), i = 1 \dots M \tag{2.2}$$

2.1.2. Planificación de sistemas de tiempo real

El cometido principal de un sistema de tiempo real es gestionar los recursos entre los procesos o tareas que los requieran, consiguiendo siempre la mínima utilización posible dentro de los requisitos de tiempo demandados. Para conseguir tal fin, se pone a prueba el sistema, afirmando la existencia de errores, pero no la ausencia total de los mismos. Para comprobar que un sistema de tiempo real es correcto y cumple las condiciones temporales requeridas, aparece la Teoría de Planificación [50], aportando la base teórica necesaria para ratificar dichas condiciones.

Se conoce como *algoritmo de planificación* al conjunto de pautas que, usando el mínimo de recursos, determina las decisiones que se han de tomar a lo largo de la vida del sistema de manera que todos los procesos cumplan los requisitos de tiempo o de cualquier otra naturaleza, como pueden ser las restricciones de precedencia. El término *planificación* se refiere a la capacidad de seleccionar una tarea en particular para su procesamiento en un instante de tiempo concreto.

La base de la planificación se divide en dos partes diferenciadas: por un lado, comprobar si el sistema es planificable, es decir, comprobar si existe una planificación dadas unas tareas con sus condiciones temporales y un determinado algoritmo, y por otro lado, encontrar dicha planificación. La primera parte se conoce como *análisis de planificabilidad*, y la segunda, como *construcción de la planificación*.

El análisis de planificabilidad para un algoritmo específico se realiza mediante pruebas (o tests), las cuales ofrecen unos resultados que reflejan la competencia de dicho algoritmo para encontrar una planificación realizable dado un conjunto de tareas. No obstante, la fiabilidad de estos tests depende, en un porcentaje alto, del coste computacional que conlleva. Se pueden distinguir tres tipos [61]:

- **Tests exactos.** Tienen resultado positivo siempre que exista una planificación, y negativo en caso contrario. Sin embargo, su complejidad computacional es alta e intratable [29], lo que conlleva una reducción de la aplicación a fases off-line.

- **Tests suficientes.** Un resultado positivo asegura la existencia de una planificación factible, pero uno negativo no asegura que no la haya. En consecuencia, este tipo de tests, a pesar de ser mas sencillos de implementar que los exactos, puede rechazar conjuntos de tareas planificables.

- **Tests necesarios.** Con este tipo de test se garantiza la imposibilidad de la planificación con un resultado negativo, pero un resultado positivo puede confirmar que exista una planificación para las tareas dadas y ésta no ser viable.

De entre todas las pruebas realizables para hacer un análisis de planificabilidad, existen dos, basadas en conceptos diferentes [44], que se usan para sistemas de tiempo real:

- **Límite de utilización.** Se entiende por utilización al porcentaje de ocupación del procesador. Cada tarea periódica τ_i carga el procesador con una utilización, que viene dada por su WCET y su periodo, y de valor menor a la unidad:
$$U_i = \frac{C_i}{T_i} \leq 1 \tag{2.3}$$
- **Tiempo de respuesta.** Se ocupa del cálculo de los tiempos de respuesta en el peor caso de todas las tareas, que el sistema comprará a posteriori con sus correspondientes plazos.

Por otro lado, los algoritmos de planificación se clasifican como estáticos o dinámicos (ver 2.1). Los algoritmos *estáticos* son conocidos como *off-line*, y son los que realizan toda la planificación con antelación a la ejecución del sistema; los *dinámicos*, también algoritmos *on-line*, sin embargo, pueden tomar decisiones en tiempo de ejecución sobre el momento en que una tarea pasa al procesador. En estos últimos, y dependiendo del momento en que las prioridades de las tareas sean establecidas, podemos referirnos a algoritmos dinámicos de prioridades estáticas (o fijas) o de prioridades dinámicas.

Los algoritmos dinámicos pueden ser, a su vez, *con apropiación* o *sin apropiación*. Esta clasificación viene del momento en que una tarea con cierta prioridad debe abandonar el procesador en favor de otra con mayor prioridad. Si en ese momento, la decisión se aplaza hasta la ejecución íntegra de las tareas, el algoritmo será sin apropiación del procesador. Sin embargo, si una tarea dada se ve obligada a abandonar el procesador sin terminar su ejecución a causa de que otra tarea con mayor prioridad está preparada, el algoritmo será con apropiación.

2.1.3. Planificación centralizada

Tradicionalmente, las técnicas de planificación tenían en cuenta una serie de tareas que se ejecutaban en un único procesador, y que eran independientes entre sí, es decir, no dependían de otras tareas para lanzarse en el mencionado procesador. Además, no existían relaciones de exclusión de ningún tipo y en consecuencia, las tareas no se podían interrumpir a mitad de su ejecución. En una planificación centralizada, los tiempos de cambio de contexto, o incluso otros tiempos relativos al sistema son omitidos, ya sea por suponerlos nulos, o por incluirlos en los tiempos de ejecución en el peor caso.

Atendiendo al momento de decisiones de planificación	Off-line	Ejecutivo cíclico	
	On-line	Prioridades estáticas	*RMS*
			DMS
		Prioridades dinámicas	*EDF*
			LLF
Atendiendo a la posibilidad de liberación del procesador	Sin apropiación		
	Con apropiación		

Tabla 2.1: Clasificación algoritmos de planificación y ejemplos para planificación centralizada

2.1.3.1. Planificación estática

Como se define previamente, la planificación estática u *off-line* (también llamada ejecutivo cíclico) es aquélla que realiza todas las decisiones de planificación antes de la ejecución del sistema. La información necesaria se almacena en una tabla que contiene todos los tiempos relativos a cada tarea, todas ellas periódicas, incluyendo los instantes de activación y finalización. Una vez comienza la ejecución, el planificador lee la tabla como si de una lista se tratase y activa las tareas en el momento adecuado.

El planificador es sencillo, robusto y eficiente. Al saber con precisión lo que el procesador ejecuta en cada momento, se convierte en un método determinista y predecible [63]. No obstante, este procedimiento genera sistemas inflexibles y difíciles de mantener [16].

El principal problema que existe es crear el plan de ejecución del sistema, habiendo dividido los procesos en porciones de código tales que se puedan ejecutar en ranuras temporales, conociendo todas las características que deben cumplir. Este es un problema NP-Completo [1] [12].

En resumen, existen casos particulares en al resolución de una planificación en los que se puede construir dicha planificación en tiempo polinómico, pero la mayor parte de los casos se deberá hacer mediante el uso de heurísticos [15, 63, 59].

Para realizar un correcto plan de ejecución, éste ha de hacerse con un intervalo temporal igual al mínimo común múltiplo de los periodos de todas las tareas, ya que si no, la longitud de la tabla de planificación sería intratable. Sumado a todo esto, la inserción de una nueva tarea en el sistema o el cambio de algún parámetro temporal en alguna de las existentes, conllevaría un cambio radical en el plan de ejecución, teniéndolo que recalcular y llegando a dividir los procesos de una manera totalmente

[1] (Reingold et al, 1977 y Wilf, 1986). Un problema se dice que es NP-Completo si es duro y es NP (Non-Deterministic Polynomial-time), es decir, denota la colección de todos los problemas decisión los cuales tienen algoritmos de solución no determinísticos en tiempo polinomial y tiene la característica de que todo problema NP se reduce polinomialmente a él.

diferente a la anterior. Otro problema relacionado con los ejecutivos cíclicos [43] sería la complejidad de incluir tareas no periódicas, a causa del carácter estático del plan de ejecución.

2.1.3.2. Planificación dinámica

Desde la década de los 70, en el mundo de los sistemas de tiempo real se usa una aproximación más acertada, la planificación en tiempo de ejecución o planificación dinámica. En ella, el planificador asigna una prioridad a cada tarea, y elige en cada instante y a partir de estas prioridades cuál es la tarea más urgente, dejando de existir un plan de ejecución previo a la puesta en marcha del sistema. En el caso de que dichas prioridades sean asignadas antes de la ejecución del sistema y no cambien a lo largo de esta, se hablará de prioridades fijas (o estáticas); pero en caso de que estas pueden cambiar a lo largo del proceso, se hablará de prioridades dinámicas.

Existen numerosos algoritmos de planificación de tareas independientes para sistemas monoprocesador, entre los que destacan los presentados por *Liu y Layland* [42]: el método de asignación de prioridades estáticas en función del periodo de las tareas (Rate Monotonic Scheduling o RMS); y el método de asignación dinámica de mayor prioridad a la tarea con el *deadline* (plazo) más próximo (Earliest Deadline Fisrt o EDF). Lo importante de estos dos algoritmos es que son los óptimos en su clase [47]. Si dado un conjunto de tareas, un algoritmo es capaz de encontrar una planificación válida, el óptimo también lo hará.

Planificación dinámica basada en prioridades fijas. Para poder tratar analíticamente el problema de la planificación adecuada dado un conjunto de tareas de tiempo real, el trabajo de *Liu y Layland* [42] asienta sus bases en las siguientes suposiciones, que crean un modelo bastante restrictivo:

1. Todas las tareas críticas del sistema son periódicas.
2. El plazo de una tarea es igual a su periodo ($T_i = D_i$).
3. Las tareas del sistema son independientes entre sí, es decir, no mantienen relaciones de precedencia, no comparten recursos, ni se comunican.
4. El tiempo de ejecución de cada tarea es contante para esa tarea y no varia en el tiempo de ejecución, entendiendo como tiempo de ejecución el que le llevaría al procesador ejecutar esa tarea sin interrupción.
5. Las tareas no se suspenden mientras ejecutan el código correspondiente a una activación.
6. Las tareas no periódicas del sistema son especiales, son rutinas de activación o de recuperación de fallos. Además, desplazan a las tareas periódicas mientras están ejecutándose y no tienen plazos de respuesta críticos.

7. Las tareas se ejecutan con un planificador basado en prioridades y expulsivo.

Acorde con estas suposiciones, se enunció el método de planificación de prioridad a la tarea mas frecuente (RMS). Según el algoritmo RMS, las prioridades se asignan con respecto a la frecuencia de las tareas, es decir, cuanto mayor sea la frecuencia de una tarea (menor periodo), mayor será su prioridad:

$$\forall \tau_i, \tau_j \in \Pi : T_i < T_j : p_i > p_j \tag{2.4}$$

Aparte de este método de planificación, se enunció la siguiente definición:

Definición 2. 1 *Liu y Layland, 1973. El instante crítico de una tarea se define como el instante en que una activación de dicha tarea conlleve el tiempo de respuesta más largo.*

Y a partir de esta definición, se enuncia este teorema:

Teorema 2. 1 *Liu y Layland, 1973. El instante crítico para una tarea ocurre cuando su activación coincide con la activación de todas las tareas con mayor prioridad que ella.*

Según este teorema, si todas las fases de las tareas sel sistema son nulas ($\phi_i = 0 \forall i = 0 \dots n$), el instante crítico para todas las tareas del sistema tendrá lugar durante el inicio del mismo.

De la misma manera, afirmar que una planificación es válida implica que un conjunto de tareas cumple sus plazos de respuesta bajo cualquier circunstancia, asegurando la planificabilidad en el caso más desfavorable, es decir, en el instante inicial del sistema, que es cuando se produce el instante crítico de todas las tareas a la vez. Todo esto viene dado por el teorema:

Teorema 2. 2 *Liu y Layland, 1973. Si existe una asignación de prioridades a un conjunto de tareas con las características enumeradas en las suposiciones previas que produzca una planificación admisible, la planificación de prioridades RMS también será admisible.*

El test RMS propuesto en el trabajo de *Liu y Layland* se fundamenta en la utilización del procesador. Este factor de utilización es definido como la fracción de tiempo que tarda la tarea en ejecutarse en el procesador. Como se definió antes, $\frac{C_i}{T_i}$ es la fracción de tiempo que tarda en ejecutarse una tarea τ_i en el procesador, por lo tanto, para un conjunto de n tareas, el factor de utilización será:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{2.5}$$

A pesar de poder incrementar el valor del factor de utilización, ya sea aumentando el valor de los C_i o disminuyendo el valor de los T_i, se verá limitado a causa de que todas las tareas tienen que satisfacer sus plazos en los momentos críticos. Para conocer más acerca de si el sistema es planificable, se analiza este factor de utilización, de donde se puede deducir que no tiene interés conocer el valor más pequeño que puede tomar, pero si es de gran importancia conocer el valor máximo.

Como se trabaja con asignaciones basadas en prioridades, se dice que un conjunto de tareas hará un uso completo del procesador si las prioridades asignadas son viables para el conjunto y si un incremento en los tiempos de computación de alguna de las tareas de dicho conjunto hiciese las asignaciones de prioridades inviables. Para un algoritmo que trabaja con conjuntos de tareas basadas en prioridades fijas, la cota máxima mas pequeña posible del factor de utilización es el mínimo de los factores de utilización sobre todos los conjuntos de tareas que llenan el procesador.

Dado un conjunto de tareas, se pueden realizar diferentes asignaciones de prioridades. El factor de utilización conseguido la asignación de prioridades lograda por la planificación *rate - monotonic* es igual o mayor al factor conseguido por cualquiera de las otras asignaciones de prioridades. Esto es debido a que la asignación lograda por la planificación *rate - monotonic* es óptima. Por consiguiente, el factor de utilización máximo en el peor caso que debe ser calculado es el mas bajo de los factores correspondientes a la asignación *rate - monotonic* sobre todos los posibles periodos y tiempos de ejecución requeridos para las tareas.

Teorema 2. 3 *Sea* $\Pi = \{\tau_i, i = 1 \dots N\}$ *un conjunto de tareas con prioridades, entonces el conjunto es planificable, para cualquier relación de fases, si:*

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{\frac{1}{N}} - 1) \tag{2.6}$$

Como se demuestra, el conjunto de tareas es planificable si la utilización del procesador es menor que un determinado umbral. Dicho umbral viene dado por:

$$U \leq N(2^{\frac{1}{N}} - 1) \tag{2.7}$$

Se puede observar, que en el caso particular de dos tareas, el mencionado umbral será igual a 0.8383:

$$U \leq 2(2^{\frac{1}{2}} - 1) = 0{,}8383 \tag{2.8}$$

y en caso de una gran cantidad de tareas:

$$\lim_{N \to \infty} N(2^{\frac{1}{N}} - 1) = ln2 \simeq 0{,}69 \tag{2.9}$$

Hay que tener en cuenta que este test, basado en utilización, es suficiente pero no necesario, ya que pueden existir conjuntos de tareas planificables que fallen la condición anterior.

En 1989, *J. Lehoczky*, *L. Sha* y *Y. Ding [39]* presentan una técnica para realizar un análisis exacto de una planificación realizado con el algoritmo RMS, ya que, hasta entonces, dicho algoritmo tenía una usabilidad reducida a causa de que las suposiciones en las que se basase hacían de éste un modelo de sistema poco realista. La nueva técnica presentada afirma que el tiempo transcurrido entre la activación

de la tarea y su finalización debe ser mayor que que la carga solicitada al procesador para dicha tarea. Como no era viable comprobar en todos los instantes temporales, se llegó a la resolución de que solo había que comprobar en los *puntos de planificación*, que son los instantes temporales en los que tareas de mayor o igual prioridad son activadas.

Teorema 2. 4 *Lechoczky et al, 1989. Sea* $\Pi = \{\tau_i, i = 1 \dots n\}$ *un conjunto de tareas ordenadas en orden decreciente de prioridades. El conjunto de puntos de planificación para una tarea* $\tau_i \in \Pi$ *coincidirá con los momentos en los que las tareas de mayor prioridad que ella son activadas y se define como:*

$$S_i = \left\{ kT_i \mid j = 1 \dots i, k = 1 \dots \left\lfloor \frac{T_i}{Tj} \right\rfloor \right\} \tag{2.10}$$

Lo que nos lleva a afirmar que la tarea cumplirá todos los plazos de respuesta para cualquier relación de fases si :

$$\min_{t \in S_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lfloor \frac{t}{T_j} \right\rfloor \leq 1 \tag{2.11}$$

Y por consiguiente, la utilización en el peor de los casos será:

$$U_i(t) = \frac{\sum_{j=1}^{i} \left(Cj \left\lceil \frac{t}{T_j} \right\rceil \right)}{t} \tag{2.12}$$

Y finalmente, podemos reescribir la ecuación 2.12 como:

$$\min_{t \in S_i} U_i(t) \leq 1 \tag{2.13}$$

A partir de esto se deduce que, para que una tarea τ_i sea planificable, el tiempo de ejecución de tareas de la misma o mayor prioridad debe ser menor que el tiempo transcurrido hasta ese instante. Si y sólo si se cumple esa condición para todas las tareas, el sistema será planificable:

$$\max_{\forall i = [1 \dots n]} \left\{ \min_{t \in S_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \right\} \leq 1 \tag{2.14}$$

La inecuación 2.14 puede ser aplicada en un caso real, ya que el algoritmo realiza la condición de comprobación en los puntos de planificación de cada tarea.

Como se comento anteriormente, las suposiciones en las que se basa RMS provocan un sistema con usabilidad reducida que hace de este un modelo de sistema poco realista. Además, es reseñable que la restricción de que el plazo de las tareas debe ser igual a su periodo ($D_i = T_i \forall i$) es muy severa. Un ejemplo de sistema que no cumple esta condición es un sistema de muestreo, en el que es importante que cada muestra este separada el periodo de muestreo, siendo el plazo de finalización el margen de tolerancia con

respecto al instante de toma de cada muestra. Puesto que en este tipo de sistemas ya no es aplicable el método RMS, surgió la necesidad de investigar sobre nuevas técnicas. En consecuencia, *Leung y Whitehead [40]* definieron el método de prioridad a la tarea mas urgente (Deadline Monotonic Scheduling o DMS), el cual queda definido en este teorema:

Teorema 2. 5 *Leung y Whitehead, 1982. Si existe una planificación de prioridades de un conjunto de tareas con las características definidas en el método RMS que produzca una planificación admisible, entonces la asignación de prioridades mayores a las tareas con plazos de respuesta menores, es decir, mas urgentes, también será admisible.*

Se puede deducir a partir del teorema que RMS es un caso particular del método DMS, cuando $D_i = T_i$. Además, en el trabajo de *Leung y Whitehead*, queda demostrado que si los plazos son iguales o menores a los periodos ($D_i \leq T_i \forall i$), la asignación DMS será óptima. Asentando todo esto como base, *Audsley [4]* desarrollo un test de planificabilidad suficiente basado en utilizaciones para DMS:

Teorema 2. 6 *Audsley, 1991. Sea* $\Pi = \tau_i, i = 1 \ldots N$ *un conjunto de tareas ordenadas en orden decreciente de prioridades, entonces la interferencia que la tarea* τ_i *experimentará debido a tareas con igual o mayor prioridad que ella estará acotada por la expresión:*

$$I_i = \sum_{j \in h_p(i)} \left\lceil \frac{D_i}{T_j} \right\rceil C_j \tag{2.15}$$

donde $h_p(i)$ es el conjunto de tareas de igual o mayor prioridad que τ_i, y que por lo tanto pueden interferir en al ejecución de esta.

El conjunto de tareas será planificable si cada una de ellas tiene una utilización máxima definida dentro de su plazo, y menor al 100 %:

$$\frac{C_i}{D_i} + \frac{I_i}{D_i} \leq 1, \forall i = 1 \ldots N \tag{2.16}$$

Un análisis exacto se obtiene extendiendo el test desarrollado por *Lehoczky et al [39]* (4) al caso DMS, redefiniendo la utilización en el peor caso (2.12) como:

$$U_i(t) = \frac{C_i \left\lceil \frac{t}{T_i} \right\rceil + \sum_{j=1}^{i} \left(C_j \left\lceil \frac{t}{T_j} \right\rceil \right)}{t} \tag{2.17}$$

Existen varios métodos para calcular el tiempo de respuesta en el peor caso (WCRT, *Worst Case Response Time*). Las técnicas que desarrollaron *Joseph y Pandya* [36]y *Audsley et al [4]* en sendos estudios permiten conjuntos de tareas con plazos menores o iguales a sus respectivos periodos ($D_i \leq T_i$), y con una asignación de prioridades cualquiera. Ambos métodos son semejantes al que desarrollo *Harter* [32, 33], en 1984, que empleó un método creado por él mismo, el Algoritmo de Dilatación Temporal (*TDA, Time*

Dilation Algorithm), que consiste en usar una lógica temporal en el calculo de los tiempos de respuesta en el peor caso, siendo pionero, además, en el uso de este parámetro.

El peor tiempo de respuesta, R_i, de una tarea τ_i es el menor $w \geq 0$ tal que:

$$w = C_i + \sum_{\tau_j \in h_p(\tau_i)} \left\lceil \frac{w}{T_j} \right\rceil C_j \tag{2.18}$$

El sistema es planificable si existe solución para la ecuación anterior (2.18), si $\exists R_i \forall i = 1 \ldots N$, y si $R_i < D_i \forall i = 1 \ldots N$. Esta ecuación, solo tiene solución si se trata de resolver de manera iterativa [7, 8]:

$$\begin{cases} w^0 = C_i \\ w^{n+1} = C_i + \sum_{\tau_j \in (\tau_i)} \left\lceil \frac{w^n}{T_j} \right\rceil C_j \end{cases} \tag{2.19}$$

Planificación dinámica basada en prioridades dinámicas. En el mismo trabajo que RMS, *C. Liu y J. Layland* presentaron varios algoritmos diferentes [42]. El algoritmo DDS (Deadline Driven Scheduling), más conocido como EDF (Earliest Deadline First), se define bajo las mismas suposiciones que RMS. EDF asigna las prioridades en base a la proximidad de los plazos de finalización, de manera que la tarea que se selecciona para ejecutarse es la que tiene más cercano la finalización de su plazo:

$$\forall \tau_i, \tau_j \in \Pi : d_i < d_j : p_i > p_j \tag{2.20}$$

Como se explica más arriba, el algoritmo asigna las prioridades en base a sus plazos. A una tarea dada se le asignará una prioridad más alta si su plazo es de valor más pequeño, es decir, más cercano en el tiempo; y se le asignará una prioridad más baja en caso contrario. Sea cual sea el instante de tiempo que se encuentre la ejecución del sistema, la tarea con mayor prioridad será la próxima en ser procesada. Este método dinámico de asignar las prioridades contrasta con RMS, el cual asignaba las prioridades y estas no cambiaban a lo largo de la ejecución del sistema.

Teorema 2. 7 *Liu y Layland, 1973. Cuando el algoritmo EDF es usado para planificar un conjunto de tareas en un procesador, no hay tiempo de procesador sin utilizar previo a un overflow.*

Este último teorema se utiliza para establecer este otro teorema:

Teorema 2. 8 *Liu y Layland, 1973. Para un conjunto dado de m tareas, el algoritmo EDF es planificable si y solo si:*

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_m}{T_m} \leq 1 \tag{2.21}$$

que puede ser expresado de la siguiente forma:

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1 \tag{2.22}$$

Para concluir, es destacable mencionar que si un conjunto tareas es planificable por cualquier otro algoritmo, entonces es planificable también por EDF, lo que lo hace óptimo. Este hecho fue intuido por *Liu y Layland* en su artículo [42], pero no será demostrado hasta 1974, por *Dertouzos* [27].

Años más tarde, *A. Mok y M. Dertouzos* propusieron un nuevo algoritmo, también considerado óptimo, que asignaba la prioridad a la menor holgura (Least Laxity First o LLF). La mencionada holgura en una tarea viene dada como la diferencia entre su plazo absoluto y el tiempo estimado de finalización en el peor caso. Los resultados que ofrece LLF son semejantes a los conseguidos con EDF, pero con una mayor sobrecarga en tiempo de ejecución, debidos a los cambios de contexto provocados por los cambios de holgura de las tareas durante la ejecución del sistema. Este fue el motivo por el que la comunidad de tiempo real se centró en EDF, amortiguando las condiciones iniciales propuestas por *Liu y Layland* por otras menos restrictivas y extendiendo los análisis a casos más generales.

Destaca también el análisis de demanda de procesador [10, 11](PDA, Processor Demand Analysis); la planificación de tareas aperiódicas basándose en estructuras denominadas servidores (como el método Total Bandwidth Server [52, 53] o TBS, o el método Constant Bandwidth Server [1, 2] o CBS); técnicas para la correcta gestión de las sobrecargas en tiempo de ejecución (como el algoritmo Capacity Sharing [21] o CASH, el algoritmo Greedy Reclamation of Unused Bandwidth [41] o GRUB, o la planificación elástica [18, 19, 20]); etc. En el articulo [48] puede encontrarse un resumen de las técnicas de análisis de planificabilidad para algoritmos de planificación basados en prioridades dinámicas.

Test de planificabilidad Rub. Este proyecto usa una forma alternativa para calcular una cota superior al tiempo de respuesta de computación lineal. Esta alternativa es un test de planificabilidad RUB, y sustituye a los clásicos test de planificación RTA. El tiempo de respuesta obtenido se considera un test suficiente, ya que en el caso del tiempo conseguido con los métodos anteriores se obtiene un tiempo exacto, mientras que este nuevo tiempo de respuesta es sólo una cota superior. Esto es debido a que los otros métodos calculaban el tiempo de respuesta a partir del número de tareas, por lo que, a pesar de tardar un mayor tiempo en converger, concluyen con tiempo de respuesta exacto. Sin embargo, RUB consigue, en un tiempo mucho menor, una cota superior aproximada, es decir, un tiempo de respuesta mayor que el exacto. Con esto se consigue saber si un conjunto de tareas dado en un nodo es planificable o no. Cabe destacar que este método no puede usarse de extremo a extremo porque los tiempos no son exactos.

Puede darse el caso de que, al tratarse de un test suficiente, el resultado sea un conjunto de tareas no planificable, pero que en realidad si que sea planificable. Estos casos serán denominados *Falso Negativo.*

El cálculo del tiempo de respuesta de dado por el test de planificabilidad RUB viene dado por la siguiente fórmula, teniendo en cuenta todas las tareas del nodo en cuestión de mayor a menor prioridad:

$$R_{rub_i} = \frac{C_i + \sum_{j=1}^{i-1} C_j \left(1 - U_j\right)}{1 - \sum_{j=1}^{i-1} U_j} \tag{2.23}$$

donde:

C_i es el tiempo de ejecución en el peor caso de la tarea τ_i.

U_j es la utilización de la tarea inmediata de mayor prioridad, es decir, $p_j > p_i$.

2.1.3.3. Plazos superiores a los periodos

Es bastante común en algunos sistemas distribuidos que los plazos de las tareas sean de un valor superior a sus respectivos periodos, donde una activación en respuesta a un evento debe pasar por distintos recursos hasta su finalización. Por eso, en estos casos hay que tener especial atención. El análisis del primer instante crítico, como el análisis realizado por *Audsley et al [6]* resultan insuficientes, ya que el tiempo de respuesta en el peor caso puede verse modificado en instantes críticos posteriores por una activación anterior de la tarea. Además debemos usar heurísticos en lugar de RMS o EDF para la asignación de prioridades, ya que éstos dejan de ser métodos óptimos [5]. *Lehoczky [38]* demostró, en el año 1990, una solución posible al problema que se plantea, extendiendo el análisis del instante crítico a todas las activaciones de una tarea que ocurran dentro de su periodo de ocupación en el peor caso. Basándose en esta solución y en el método propuesto por *Audsley, Tindell et al [57, 60]* propusieron una nueva técnica, exacta, basada en la obtención de los tiempos de respuesta, independientemente de la asignación de prioridades, y adecuada para cualquier relación plazo-periodo de las tareas:

$$w_i^{n+1}(p) = B_i + (p+1)C_i + \sum_{\tau_j \in h_p(\tau_i)} \left\lceil \frac{w_i(p)^n + J_i}{T_j} \right\rceil C_j \tag{2.24}$$

De nuevo, para resolver la ecuación ha de hacerse de manera iterativa, iniciando cada iteración sobre p con el valor $w_i^{(0)}(p) = (p+1)C_i$, y deteniendo su análisis cuando $w_i(p) \leq (p+1)T_i$.

Por tanto, el tiempo de respuesta en el peor caso para la tarea τ_i, será el máximo de los tiempos de respuesta obtenidos:

$$R_i = \max_{\forall p} \{w_i(p) + J_i - qT_i\} \tag{2.25}$$

y se podrá concluir que el conjunto de tareas es planificable si todos los tiempos de respuesta en el peor caso son menores que sus plazos.

2.2. Sistemas distribuidos

Tradicionalmente se trabajaba con un solo procesador, pero el avance de las tecnologías hizo que los procesadores fueran mas eficientes, además de poder integrar varios en un mismo sistema y delegarle una

funcionalidad diferente a cada uno. En resumen, se pasa de tener una arquitectura centralizada, a tener una arquitectura distribuida, más compleja, y de la que se puede afirmar que es más fiable.

Según Coulouris [24] se define un sistema distribuido como:

Definición 2. 2 *Definición 2.5. Un sistema distribuido es una colección de ordenadores autónomos conectados por una red de computación y equipados con software distribuido. El software distribuido permite a los computadores coordinar sus actividades y compartir los recursos del sistema - hardware, software y datos. Los usuarios de un sistema distribuido bien diseñado deben percibir una facilidad de computación simple e integrada aunque pueda estar implementada por muchos computadores en diferentes localizaciones.*

Por lo que se puede definir un sistema distribuido [9] como un conjunto de entidades autónomas que cooperan entre sí para alcanzar un fin común. Para este propósito, deben intercambiar información mediante el uso de mensajes, necesitando un sistema de comunicaciones. A partir de esta definición, se puede afirmar que en el caso de estudio de este proyecto, un sistema de tiempo real distribuido consiste en un sistema de tiempo real en el que intervienen una serie de condiciones y requisitos de tiempo real.

En [24], *Coulouris* fija en seis las características principales que son responsables de la utilidad de los sistemas distribuidos: compartición de recursos, apertura, concurrencia, estabilidad, tolerancia a fallos y transparencia.

En resumen, la planificación de sistemas distribuidos acarrea una serie de problemas adicionales frente a la planificación de sistemas centralizados, como la planificación de los canales de comunicaciones y y la asignación de tareas a los distintos procesadores. Por consiguiente, se puede afirmar que un sistema de tiempo real distribuido está compuesto por uno o más procesadores que interactúan con el entorno, conectados entre sí por mediante un bus o una red de comunicaciones. Estas redes de comunicaciones son las que ponen el problema añadido de tener que planificar los canales, y los distintos procesadores los que agravan la asignación previa de tareas.

Al margen de estas características, destacan dos enfoques a la hora de desarrollar los sistemas distribuidos [37]:

- Sistemas gobernados por eventos. Los procesos, tareas o acciones son activadas por ocurrencia de eventos, es decir, instantes temporales que no son conocidos en el momento de diseño del sistema, y que caracterizan el flujo del programa.

- Sistemas gobernados por tiempo. Las acciones o procesos son activados en instantes de tiempo predeterminados, habitualmente de forma cíclica y a intervalos de tiempo regulares. Los eventos de activación de las tareas son todos periódicos e independientes. Además, los tiempos exactos de activación (tiempo de transacción y acciones que los componen) son conocidos en la fase de diseño del sistema, previa a la ejecución. Por tanto, el único evento que activa las operaciones es el reloj interno del sistema.

De lo anterior se deduce que en un sistema gobernado por eventos existe una dependencia muy estrecha entre acciones de una misma transacción, ya que un cambio en cualquier tarea o mensaje afecta directamente a todos los demás procesos.

Sin embargo, un sistema gobernado por tiempo requiere que los nodos y la red estén sincronizados, lo que implica una base de tiempos global. De esto se desprende que los desplazamientos temporales de las acciones son mayores que los tiempos de respuesta en el peor caso de las predecesoras. En consecuencia, la planificación de cada recurso (nodo o red) es independiente, lo que evita realizar un análisis de planificabilidad iterativo.

Asimismo, se han de caracterizar los eventos que disparan la ejecución de las acciones. Por ejemplo, un sensor que mide algún parámetro en un sistema; llegado a cierto nivel de medición, el sensor requiere que el sistema se encargue de procesar los datos recogidos hasta el momento, por lo que origina un evento que activa dicho procesamiento. Una clasificación comúnmente aceptada es la basada en la fuente que origina el evento [46]:

- Eventos temporizados. Producidos por los relojes internos del sistema o por temporizadores programados.
- Eventos externos. Originados en el propio sistema físico que se está controlando. Se desencadenan una serie de acciones como respuesta a ese evento.
- Eventos internos. Producidos dentro del sistema, necesarios para la correcta sincronización del mismo.

2.2.1. Planificación distribuida

Hoy en día, la planificación y el análisis en sistemas distribuidos y multiprocesador sigue siendo una asignatura pendiente en cuanto a la obtención de resultados y conclusiones. Al contrario de lo visto en el apartado de los sistemas monoprocesador, donde se contaba con gran cantidad de referencias y trabajos de diferentes expertos, los sistemas distribuidos tienen gran cantidad de lineas abiertas a la investigación, en lo que se refiere a la búsqueda de soluciones mejores y más genéricas, para poderlas aplicar a un mayor número de sistemas de este tipo. Se ha demostrado [28, 55] que muchos problemas de planificación de tiempo real en sistemas multiprocesadores son NP-Completos.

Al tratar de obtener soluciones a los problemas mencionados anteriormente y evaluar los resultados, se puede comprobar que los sistemas con varios procesadores y distribuidos tienen una alta complejidad. De la misma forma, las conclusiones sacadas para sistemas centralizados monoprocesador no se pueden dar por supuesto para sistemas distribuidos, ya que serían contraproducentes. Esto se ve reflejado en el estudio de *Stankovic et al* [55], en el que tenemos varios ejemplos: la apropiación del procesador no siempre es beneficiosa como en sistemas centralizados; el uso de un algoritmo de planificación dinámica basada en

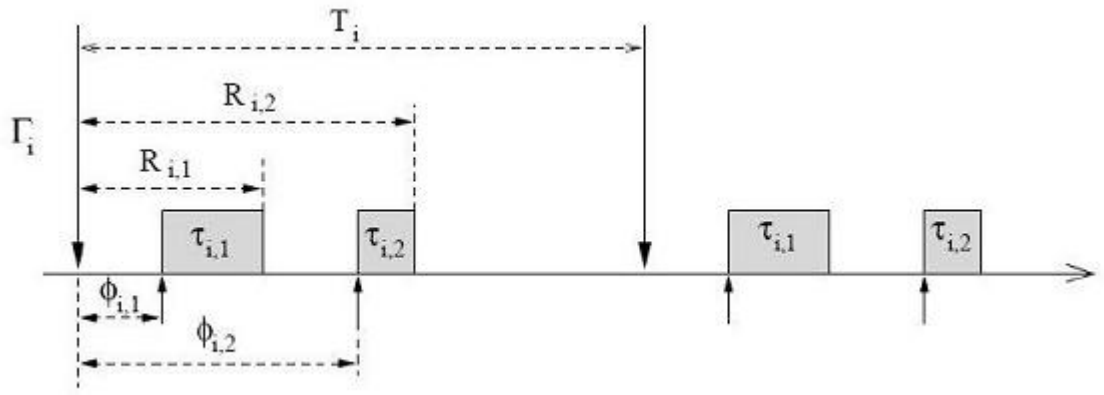

Figura 2.1: Ejemplo de transacción de tiempo real de [9]

plazos no es óptimo sin el conocimiento completo del sistema; el hecho de aumentar el número de procesadores puede causar que un conjunto de tareas dado deje de ser planificable. Todas estas características conducen a que se haya recurrido al uso de heurísticos para la planificación de tiempo real distribuido, tales como el templado simulado de *Tindell* [59], el algoritmo HOPA [30] (Heuristic Optimized Priority Assignment), o los algoritmos WCDO (Worst Case Analysis for Dynamic Osets) [45] y WCDOPS (Worst Case Analysis for Dynamic Osets with Priority Schemes)[46].

2.2.1.1. Modelo y política de planificación para el análisis de sistemas distribuidos

Aunque existen diferentes métodos de planificación de un sistema distribuido, en este documento sólo se va a desarrollar el que se va a usar para la finalización de este proyecto fin de carrera. En este modelo, la red se trata como un recurso más sobre el que se ejecutan acciones; en este caso particular, la transmisión de mensajes.

El modelo que se ha utilizado es una variante del **Modelo transaccional de tiempo real**. Se basa principalmente en la definición de transacciones de tiempo real. Una transacción (figura 2.1), es una entidad activada por un evento externo periódico, de período T_i, que agrupa tareas con período idéntico al evento externo, y cuyos instantes de activación están relacionados.

donde se define transacción como un conjunto de tareas ordenado según si instante de activación:

$$\Gamma_i = \{\tau_{i,j}(C_{i,j}, D_{i,j}, T_i, \phi_{i,j}, J_{i,j}, p_{i,j}), j = 1 \ldots N\} \tag{2.26}$$

donde $\phi_{i,j}$ es es el desplazamiento temporal u *offset*, $J_{i,j}$ representa el retraso máximo o *jitter*. De esta forma, la activación de cada tarea viene dada en el intervalo $[t_0 + \phi_{i,j} - J_{i,j}, t_0 + \phi_{i,j} + J_{i,j}]$, donde t_0 es el instante de llegada del evento externo. El tiempo de respuesta de una tarea viene dado por la diferencia

de tiempo entre su activación, y su finalización. Se deduce que su activación coincide con la llegada del evento externo, es decir, t_0. Por otro lado, se denota $R_{i,j}$ como el tiempo de respuesta en el peor caso. Este modelo es utilizado por diversas políticas de planificación de tareas en sistemas de tiempo real distribuido. Entre ellas destacan el algoritmo de Modificación de Fase [13], el análisis holístico de *Tindell [58, 61]* y el protocolo Release Guard Protocol [56].

La política utilizada para realizar este proyecto es una variante del *Algoritmo de Modificación de Fase*, propuesta por *M. Calha* [22], que reduce el efecto de las activaciones retrasadas. Una tarea dada es activada en respuesta a un evento, un tiempo determinado después de la llegada éste, teniendo en cuenta el tiempo de respuesta en el peor caso de las tareas predecesoras, y cerciorándose de que éstas hayan terminado en el momento de la activación de la tarea en cuestión. Además, se requiere la sincronización entre los distintos procesadores involucrados en una transacción.

2.2.1.2. Análisis de planificabilidad

Para hacer un análisis de planificabilidad del sistema de manera correcta hay que tener en cuenta más cosas de las que se tenían en consideración en un sistema monoprocesador, ya que se debe pensar en los tiempos de transmisión de la red y la procedencia de las tareas, y no sólo la ubicación de estas en cada procesador. Los distintos análisis que se realizan son:

Análisis de la red de comunicaciones. Para hacer un buen análisis de planificabilidad de una transacción o secuencia de acciones, no basta con conocer los tiempos de ejecución en el peor caso de las tareas, si no que se debe saber los tiempos de respuesta en el peor caso para la transmisión de los mensajes que se usan para comunicar las mencionadas tareas de la transacción. Asimismo, el análisis que se emplea habitualmente para la planificación de red supone que los mensajes son enviados por la red en paquetes de tamaño fijo, fragmentándose en caso necesario, y con prioridades asignadas para su correcta transmisión.

Efecto del retraso en un sistema distribuido. Como se ha comentado al comienzo de este apartado, a la hora de planificar un sistema distribuido, hay que pensar en la procedencia en el momento de la activación de las tareas y los mensajes que comunican entre sí las tareas existentes en una transacción. *Tindell [57]* propuso un modelo afirma que el instante de activación de una tarea, τ_j, viene dado por el instante de finalización de la tarea precedente. Aunque el *jitter* de la tarea inicial sea nulo, $J_1 = 0$, los procesos posteriores se ven afectados hasta un tiempo máximo de valor igual a la máxima variación del tiempo de respuesta del proceso predecesor.

Planificación holística bajo la aproximación de tareas independientes. Esta técnica fue desarrollada por *Tindell y Clark* en su trabajo [57] para su uso en sistemas distribuidos. El método consiste en calcular el tiempo de respuesta en el peor caso para cada acción que se vaya a procesar, aplicando el mismo procedimiento que para sistemas monoprocesador [60] que fue expuesto en el apartado 2.1.3.3, teniendo

en cuenta el recurso donde se ejecuta la acción, ya sea un nodo o la red. Esta aproximación hace una estimación más pesimista de los tiempos de respuesta a causa de que considera las tareas independientes entre sí y, es decir, no toma en consideración las relaciones de precedencia de las tareas implicadas. Por tanto, a la hora de definir el instante de activación de una tarea, τ_j, se calcula igual que en los sistemas monoprocesador (ecuación 2.24):

$$w_i^{n+1}(p) = B_i + (p+1)C_i + \sum_{\tau_j \in h_p(\tau_j)} \left\lceil \frac{w_i^n(p) + J_i}{T_j} \right\rceil C_j \tag{2.27}$$

Por lo que el tiempo de respuesta en el peor caso de la tarea τ_i será el máximo de los tiempos de respuesta obtenidos:

$$R_i = \max_{\forall p} \{w_i(p) + R_{i-1} - qT_i\} \tag{2.28}$$

2.3. Paradigma de servicios

Como se introdujo en el primer capítulo, un servicio es, por definición [35]: *un dominio de control de una envergadura acotada que contiene un conjunto de tareas que cooperan para alcanzar objetivos relacionados*, es decir, una entidad software, independiente de otros servicios, desarrollada, implementada y efectuada de manera totalmente individual, con sus interfaces bien definidas y que provee una determinada funcionalidad. Además, es *sin estado*.

Esta funcionalidad que puede tomar cada servicio, hace que estos aporten un valor añadido y su especificación viene dada por las características intrínsecas del entorno en el que dicho servicio va a ser utilizado. Por ejemplo, no es necesario tener medidas de seguridad en una interfaz de un servicio de información del tráfico, pero si habría que tener dichas medidas en un servicio de compra-venta por Internet. Por otro lado, una cualidad que debe tener a toda costa un servicio es que sus posibles implementaciones no estén acotadas para una determinada plataforma o lenguaje. Lo ideal es que varios servicios implementados en distintos lenguajes puedan comunicarse a través de sus interfaces.

La figura 2.2 representa el modelo conceptual de una arquitectura orientada a servicios. De la figura se desprende que los proveedores de servicios han de registrar los servicios, y que las interfaces diseñadas entre proveedores y consumidores hacen su comunicación posible con ayuda de metadatos; los metadatos incluyen funcionalidad y características como la calidad de servicio (QoS). Los consumidores, por su parte, y con ayuda de los mencionados metadatos, establecen un contrato con el proveedor del servicio requerido, y ambos están obligados a cumplir una serie de requisitos, por ejemplo, de calidad de servicio (de entrada o salida), de peticiones máximas por intervalo, etc.

La composición de servicios consiste en la combinación de las funcionalidades de servicios ya existentes para la creación de un servicio complejo, o compuesto. Para definir un servicio compuesto, que también se puede denominar aplicación, se debe tener en cuenta seis dimensiones [3]:

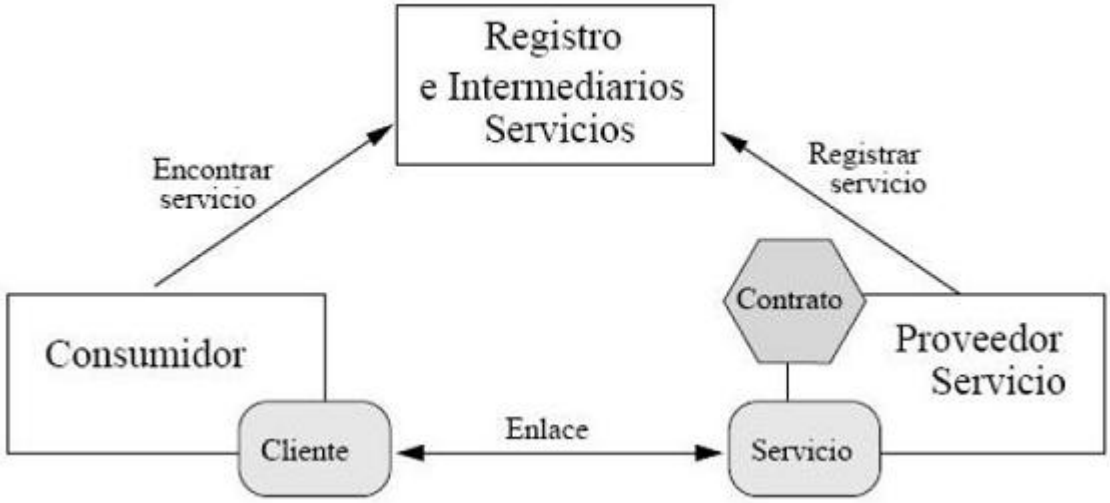

Figura 2.2: Esquema de una arquitectura orientada a servicios de [9]

- Modelo de componentes. Define la naturaleza de los elementos a ser compuestos.
- Modelo de orquestación. Explica abstracciones y lenguajes para especificar el orden en que los servicios deben ser ejecutados.
- Modelo de datos y acceso a datos. Contiene las especificaciones de los datos utilizados en la composición y cómo son intercambiados entre los distintos componentes.
- Modelo de selección de servicio. Describe cómo se seleccionan los servicios a ser utilizados como componentes, tanto estática como dinámicamente.
- Transacciones. Define las semánticas transaccionales a aplicar en la composición.
- Gestión de excepciones. Define la gestión de los errores y situaciones excepcionales ocurridas durante la ejecución de la composición.

2.3.1. Servicios Web

Al comenzar el siglo XXI, nace una tecnología que da soporte a la computación distribuida, los servicios Web [3]. Un servicio Web es [62], según el *World Wide Web Consortium* (W3C), *un sistema software diseñado para dar soporte a interacciones de máquina a máquina interoperables sobre la red. Posee una interfaz descrita en un formato procesable por una máquina (WSDL específicamente). Otros sistemas interaccionan con el servicio Web tal y como se especifica en su descripción utilizando mensajes SOAP, que*

típicamente se transmiten utilizando HTTP con serialización XML y en conjunción con otros estándares relacionados con la Web.

En resumen, se basan en la descripción de los servicios mediante el estándar WSDL, publicación de los mismos en repositorios UDDI y acceso a los servicios mediante el protocolo SOAP. Puesto que no existe un convenio sobre el uso de los servicios Web, no se puede afirmar que las tecnologías citadas son las que rigen dichos servicios.

Para obtener un servicio Web complejo hay que combinar una serie de servicios Web con distintas funcionalidades. Como es sabido, la composición de servicios complejos no es trivial [3], por lo que conviene contar con la ayuda de un middleware de composición, que provea abstracciones e infraestructuras para la definición y ejecución de los servicios compuestos.

2.4. Algoritmo de generación de tareas

Los sistemas distribuidos están compuestos por nodos con varios cometidos simultáneos. Así, en el momento de componer una aplicación, los servicios que la componen no son los únicos que se procesan en un nodo. Por tanto, para simular un sistema real, hay que tener en cuenta la utilización de los nodos generada por todas las tareas existentes de otras aplicaciones y atribuciones. Para este proyecto, se ha utilizado el algoritmo UUniFast para la generación de tareas previas de un nodo y simular su utilización previa.

El algoritmo UUniFast [14] se encarga de la generación de tareas de forma sintética dada una utilización. La finalidad de este algoritmo es generar tareas para simular la utilización previa del nodo, y que éstas tengan los valores de sus periodos según una distribución uniforme.

El algoritmo se basa en que la función de distribución de una suma de variables aleatorias independientes viene dada por la convolución de sus funciones de distribución.

Partiendo del hecho de que las variables que se usan tienen una distribución uniforme [0,1], la función de distribución acumulativa, dada por la convolución de las demás, viene expresada de la siguiente manera:

$$F_i(u) = \begin{cases} 0 & si\ u \leq 0 \\ \left(\frac{u}{b}\right)^i & si\ 0 < u \leq b \\ 1 & si\ u > b \end{cases} \tag{2.29}$$

Por consiguiente, a partir de una utilización y de un número de tareas previas, se puede hacer una repartición de dicha utilización entre las tareas existentes de forma que queden distribuidas uniformemente.

Capítulo 3

Modelos teóricos y decisiones de diseño

Este capítulo describe los modelos utilizados en el desarrollo de la tesis de Iria Estévez Ayres [9], en la cual está basado este proyecto, y expone las decisiones tomadas para el diseño que se propone. En primer lugar, se describirá el modelo de las aplicaciones basadas en servicios. Posteriormente, se realizará un estudio sobre el modelo de sistema usado y el modelo de servicios. Finalmente, se resumirán y se enumerarán las principales conclusiones de las decisiones de diseño de los modelos.

3.1. Modelo del sistema

Uno de los objetivos planteados en la tesis de *Iria Estévez Ayres [9]* era dotar de flexibilidad a las fases de diseño y ejecución de los sistemas de tiempo real distribuidos, añadiendo conceptos del campo de orientación a servicios, como puede ser el propio concepto de servicio, o el de aplicación como una composición de servicios. Se pretendía conseguir, entre otros objetivos, que un sistema tuviera la posibilidad de cambiar las implementaciones de servicio en tiempo de ejecución, sin que este hecho afectara a las prestaciones de las aplicaciones existentes en el sistema. Por ejemplo, otra finalidad de la flexibilidad era que la aparición o baja de nodos físicos, aplicaciones o implementaciones de servicio pudiera hacerse en tiempo de ejecución.

La conceptualización que se desarrolló como punto de partida es bastante utilizada en la confección de sistemas distribuidos de tiempo real, y consiste en pensar en la aplicación como un conjunto de tareas que se ejecutan en distintos nodos o procesadores, los cuales intercambian mensajes a través de una red de comunicaciones.

Para plantear un sistema un sistema de tiempo real distribuido basado en transacciones, este proyecto se centra en el diseño y análisis holístico descrito por *M. Calha [22]* y citado en el apartado 2.2.1.1 del presente documento. La aproximación tiene dos fases diferenciadas; por un lado, comprobar que se cumplen los requisitos previstos de un extremo a otro de la transacción, conociendo los parámetros de

cada tarea involucrada, como su plazo, tiempo de ejecución en el peor caso, desplazamiento o periodo; y por otro lado, calcular esos parámetros que aseguran que dichos requisitos se cumplan. En la tesis de *Estévez Ayres* se empleó un modelo holístico del sistema, ya que era necesario considerar la aplicación como un conjunto, teniendo en cuenta servicios, implementaciones de éstos y tareas previas instanciadas en cada nodo o procesador, con el fin de establecer sus parámetros.

Una vez elegido el modelo, lo siguiente a elegir es la aproximación: sistema gobernado por eventos, o sistema gobernado por tiempo. En la tesis de *Estévez Ayres* se hace una combinación de ambos modelos para obtener características de ambos enfoques: la flexibilidad y eficiencia del paradigma gobernado por eventos, y la predictibilidad y seguridad del gobernado por tiempo. Las aplicaciones existentes en el sistema estaban gobernadas por tiempo, y los instantes de activación predefinidos de las tareas eran mayores que los tiempos de respuesta en el peor caso de las tareas predecesoras. Por otro lado, se incluyó una parte más dinámica, que daba soporte para la comunicación, de forma que las señales de control estaban gobernadas por eventos; esta parte es imprescindible para proporcionar flexibilidad al sistema completo, y tener así facilidad de incluir en el sistemas nuevas aplicaciones, nuevos nodos, y nuevos servicios. De esta forma, la definición del modelo temporal de implementación de servicio define que los procesos asociados a dichas implementaciones sean activados por tiempo, mientras que las señales de control eran controladas por eventos.

Decisiones de diseño. El modelo del sistema de este proyecto se diseña excluyendo uno de los objetivos que se presentaban en la tesis de *Estévez Ayres*, concretamente el de dotar de flexibilidad a la ejecución de los sistemas de tiempo real distribuidos. Este objetivo consistía, en tiempo de ejecución, en permitir eliminar, añadir o modificar las implementaciones de servicio de una aplicación, sin que afectara a otras aplicaciones del sistema. El objetivo se ha considerado fuera de los límites de este proyecto fin de carrera, pasando a ser como una posible línea futura de trabajo, es decir, se ha prescindido de la flexibilidad ya que lo que se busca en este trabajo es la implementación de algoritmos para la composición inicial de aplicaciones basadas en servicios.

En consecuencia, se ha elegido un sistema gobernado por tiempo para las activaciones del sistema, en lugar de uno híbrido como diseñaba *Estévez Ayres*. Esto es debido a que al no tener la flexibilidad en tiempo de ejecución no hay modificaciones ni nuevas implementaciones debidas, por ejemplo, a los sensores o a nuevos nodos en el sistema, lo que implica que el sistema diseñado es predecible y periódico en todo momento.

3.2. Modelo de Aplicaciones

Como se mencionó en el primer capítulo, actualmente, los sistemas distribuidos son más dinámicos, permitiendo distribución, auto-reconguración, portabilidad y migración de aplicaciones. Asimismo, se señaló que a consecuencia de esto aparecen nuevos paradigmas de desarrollo de aplicaciones basados en el

uso de múltiples servicios esparcidos en el entorno [49, 34]. Con estos nuevos paradigmas, el desarrollo y la ejecución de las aplicaciones son más flexibles. Como caso particular, el uso del paradigma de orientación a servicios posibilita la creación de aplicaciones de forma dinámica a partir de servicios con interfaces bien especificadas que pueden ser invocados en secuencia.

Basándose en la idea de que los conceptos del paradigma de orientación a servicios pueden usarse de manera provechosa en sistemas de tiempo real, *Estévez Ayres [9]* se centra en aplicarlos en su tesis doctoral.

Las principales características destacables de las aplicaciones son:

- Aplicaciones distribuidas. Con varios procesadores aunque está permitido usar sólo un nodo físico y conseguir así una aplicación centralizada.
- Aplicaciones flexibles. Es capaz de ejecutarse en un entorno que puede estar sometido a cambios de forma dinámica.
- Aplicaciones temporalmente predecibles. Forma parte de entornos que requieren un comportamiento determinista en funcionalidad y en tiempo. El hecho de conjugar una distribución con garantías temporales implica tener que hacer un uso de comunicaciones deterministas. Además, los modelos que se presentaron en el trabajo de *Estévez Ayres*, a pesar de poder usarse en otros entornos, eran más adecuados para ser usados con aplicaciones de tiempo real no críticas.
- Aplicaciones con calidad de servicio. Dependiendo de los recursos que se tengan asignados, puede conseguirse una calidad u otra para la aplicación.

Las aplicaciones están compuestas por servicios, entendiendo por servicio una entidad software autocontenida con una funcionalidad determinada. Cada servicio puede tener una o varias implementaciones de servicio, las cuales pueden coexistir en el sistema. Las implementaciones de servicio tienen características propias, tanto temporales como de calidad de servicio, lo que implica que cada una de ellas puedan originar unos resultados totalmente diferentes y con distinta calidad.

Las implementaciones de servicio están repartidas por toda la red, en los diferentes nodos físicos, permitiendo que varias implementaciones, del mismo servicio o no, puedan compartir un mismo nodo, o varias aplicaciones puedan compartir la misma implementación. La comunicación entre las distintas implementaciones de servicio que se alojan en nodos físicos distintos se realiza mediante mensajes, regulados por los protocolos de la red subyacente; mientras que la comunicación entre implementaciones que alojadas en el mismo nodo se realiza a través de bufferes de comunicación.

La diversidad de implementaciones de servicio existente en el sistema permite a éste hacer diferentes combinaciones que cumplan los requisitos, es decir, el sistema hace una selección de implementaciones válidas, con unas determinadas características temporales, que consigue una adaptación a una métrica conocida previamente.

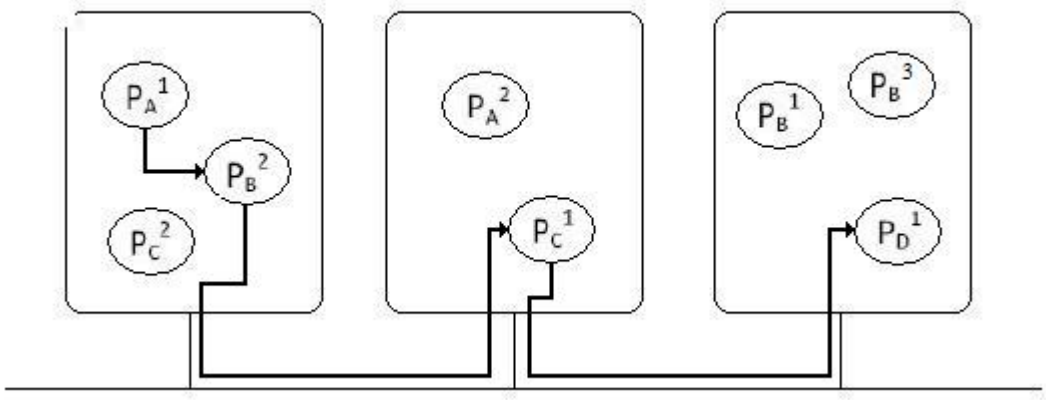

Figura 3.1: Ejemplo de aplicación distribuida

Acorde con el estudio de *Estévez Ayres* [9], un sistema de tiempo real distribuido que permita composición de aplicaciones debe facilitar:

- Comunicaciones deterministas. El protocolo de red subyacente debe garantizar tiempos acotados.
- Flexibilidad. Debe ofrecer facilidades para la instanciación y reconfiguración de las tareas o mensajes del sistema.
- Gestión de información relativa a cada implementación. Es necesario un conocimiento previo de las características de cada implementación para que el sistema, que tiene libertad para hacerlo, pueda seleccionar una combinación adecuada. Estas características deben ser proporcionadas por los desarrolladores de las implementaciones de servicio.
- Un mecanismo que realice la composición y gestione las implementaciones. Debe controlar de manera transparente la ejecución de las tareas y la transmisión de los mensajes. Asimismo, se encarga de reconfigurar las aplicaciones en entornos donde este permitida la composición dinámica.

En este tipo de entornos, la gran variedad de implementaciones de las que se dispone, permite al sistema seleccionar, en tiempo de ejecución, las que mejor se adapten a las necesidades de una aplicación concreta.

En la figura 3.1 se puede ver un ejemplo de aplicación. La aplicación, A, esta compuesta por una sucesión de servicios, $\{S_A, S_B, S_C, S_D\}$. Cada servicio está representado en el sistema por una o varias implementaciones, como se definió antes, las cuales están distribuidas en varios nodos. El servicio S_A está representado por las implementaciones $P_A = \{P_A^1, P_A^2\}$, el servicio S_B por $P_B = \{P_B^1, P_B^2, P_B^3\}$, el servicio S_C por $P_C = \{P_C^1\}$, y el servicio S_D por $P_D = \{P_D^1\}$.

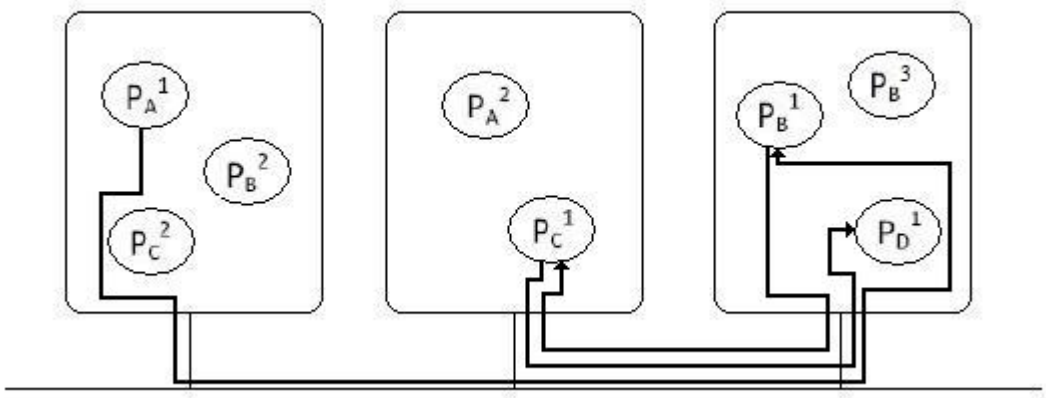

Figura 3.2: Ejemplo de una aplicación distribuida tras ser reconfigurada

Posteriormente, el sistema es el encargado de elegir las implementaciones adecuadas a las necesidades de la aplicación. La combinación resultante en el caso del ejemplo son las implementaciones $\{P_A^1, P_B^2, P_C^1, P_D^1\}$, que forman la aplicación o servicio compuesto. Cabe destacar que las implementaciones de los servicios A y B comparten el nodo físico, mientras que las implementaciones de los servicios siguientes están en otros nodos.

Por tanto, el esquema de la aplicación queda así:

$$P_A^1 \overset{msg_1}{\longrightarrow} P_B^2 \overset{msg_2}{\longrightarrow} P_C^1 \overset{msg_3}{\longrightarrow} P_D^1 \tag{3.1}$$

Si, en un momento determinado la implementación P_B^2 falla, el sistema procede a reconfigurar la aplicación para evitar un error fatal de la misma. Para ello, busca una implementación que sustituya a la que ha fallado y que siga cumpliendo las necesidades requeridas; en este caso, la implementación P_B^1. La nueva aplicación resultante sería:

$$P_A^1 \overset{msg_1^{'}}{\longrightarrow} P_B^1 \overset{msg_2^{'}}{\longrightarrow} P_C^1 \overset{msg_3^{'}}{\longrightarrow} P_D^1 \tag{3.2}$$

El modelo utilizado parte de un principio de transparencia en cuanto a las implementaciones de servicio, por lo que la ubicación de éstas no debe afectar al resto de sistema. Esto implica que si una implementación intermedia cambia por otra, no afecta a los servicios que están interactuando directamente con el que ha sufrido el cambio. Es destacable que los mensajes sí que sufren cambios, ya que la ruta seguida a través de la red no es la misma de unas implementaciones a otras.

En resumen, el modelo de servicio esta basado en una planificación holística usando flujos de datos.

Esto viene reflejado en la tesis de *M. Calha*, donde se modela la interacción entre tareas como un flujo de datos siguiendo el modelo productor-consumidor. El planteamiento de la planificación holística está diseñado en dos fases: en la primera se determinan los parámetros dependiendo del uso de recursos de interacción entre tareas, y en la segunda se realiza la planificación de cada nodo y de la red. *M. Calha* se centró en la determinación de los parámetros, definiendo varios contextos diferentes. Este proyecto se basa en uno de estos contextos, en el cual la ejecución de la tarea depende del tiempo de transmisión del mensaje que lo inicia, y, en consecuencia, del momento de inicio del mensaje en cuestión.

Todo esto, unido al Algoritmo de Modificación de Fase, implica una sincronización de los procesadores involucrados en una transacción. Como se explicó en el apartado 2.2.1.1, dicho algoritmo asegura que el instante de activación de una tarea va a producirse un tiempo fijo después de la finalización del evento que la activa, consiguiendo que cada tarea se active después de todas las que la preceden en la transacción. Para calcular el instante de activación concreto, se tiene en cuenta el tiempo de respuesta en el peor caso de las tareas previas. Cabe destacar que esto sólo es visible en un sistema gobernado por tiempo.

Decisiones de diseño. Para el modelo de aplicación se ha empleado el diseño descrito previamente, es decir, se centra en la composición de servicios heterogéneos repartidos por una red de tiempo real, en la cual cada aplicación estará compuesta por una secuencia de servicios que son invocados por la aplicación con una determinada configuración. Estos servicios se comunicarán entre sí a través de mensajes, aunque la comunicación entre dos implementaciones sea en el mismo nodo. Finalmente, una aplicación en ejecución queda definida por una combinación de implementaciones de servicio seleccionadas y los mensajes de red con los que intercambian información.

El modelo y las decisiones para el diseño de los servicio y sus respectivas implementaciones quedan reflejadas en los siguientes apartados.

3.3. Modelo de servicio

El modelo de ejecución del sistema consiste en un conjunto de servicios. Como se definió en el apartado anterior, cada servicio es una entidad software autocontenida, con una funcionalidad concreta (respuesta de sensores y actuadores, filtros para sistemas multimedia, codificadores, decodificadores, etc.). Esta funcionalidad puede ser llevada a cabo por diferentes implementaciones de servicio, no necesariamente ubicadas en el mismo procesador. Sea S la representación del sistema consistente en un conjunto de n servicios:

$$S = \{S_1, S_2, \ldots, S_n\} \tag{3.3}$$

Cada servicio, S_i, será instanciado por un conjunto de m implementaciones:

$$P_i = \{P_i^1, P_i^2, \ldots, P_i^m\} \tag{3.4}$$

Cuando se necesite utilizar un determinado servicio, se seleccionan de entre todas las implementaciones existentes en el sistema, una que cumpla los requisitos establecidos por el usuario que lo demanda o por el propio sistema. Esta decisión tiene en cuenta el estado del sistema en ese instante.

La aproximación de efectuar los servicios como implementaciones de éstos, otorga, en un entorno que permita composición dinámica, cierta tolerancia a fallos a nivel de aplicación. Una vez elegida la implementación de un servicio S_i, P_i^j, si el sistema detecta un fallo en tiempo de ejecución, éste puede seleccionar una nueva implementación, P_i^k, que, implementando la misma funcionalidad, permite que el sistema sobreviva.

Asimismo, como las prestaciones de la aplicación varían en el tiempo debido a la carga computacional que soportan los nodos físicos o procesadores, el uso de implementaciones posibilita realizar un equilibrado de carga, repartiendo la misma entre los nodos libres del sistema.

3.3.1. Implementaciones de servicio

Cada invocación a una implementación de servicio se materializa en una única tarea. Las tareas en el sistema son periódicas, de periodo T_i^t, o esporádicas, con periodo mínimo de activación $T_{min,i}^t$. Para caracterizarlas se usa un modelo comúnmente empleado en la caracterización de tareas y que se presentó en el apartado 2.1.1. Así, cada invocación periódica de la implementación P_i viene caracterizada como:

$$\tau_{i,j}^t = (C_{i,j}^t, D_{i,j}^t, T_{i,j}^t, \phi_{i,j}^t, p_{i,j}^t) \tag{3.5}$$

y cada invocación esporádica como:

$$\sigma_{i,j}^t = ((C_{i,j}^t, D_{i,j}^t, T_{min\{i,j\}}^t, p_{i,j}^t)) \tag{3.6}$$

Por otro lado, y dependiendo de su interacción con el entorno, las tareas se pueden clasificar como consumidoras, productoras, consumidoras/productoras o independientes. Si la tarea pertenece a cualquiera de los tres primeros grupos son llamadas interactivas, y esas interacciones se realizan mediante mensajes a través de la red, en caso de sistemas distribuidos, o de bufferes en caso de sistemas centralizados.

Los mensajes son considerados entidades independientes, cuya transmisión será gobernada por la red. Cada mensaje tendrá también características propias: tiempo de transmisión en el peor caso, C_i^m, un plazo relativo, D_i^m, un periodo, T_i^m, y un desplazamiento, ϕ_i^m, y vendrá definido por:

$$\tau_i^m = (C_i^m, D_i^m, num_{msg}, R_i^m) \tag{3.7}$$

donde C_i^m es el tiempo de computación del mensaje o tiempo de transmisión en el peor caso; D_i^m es

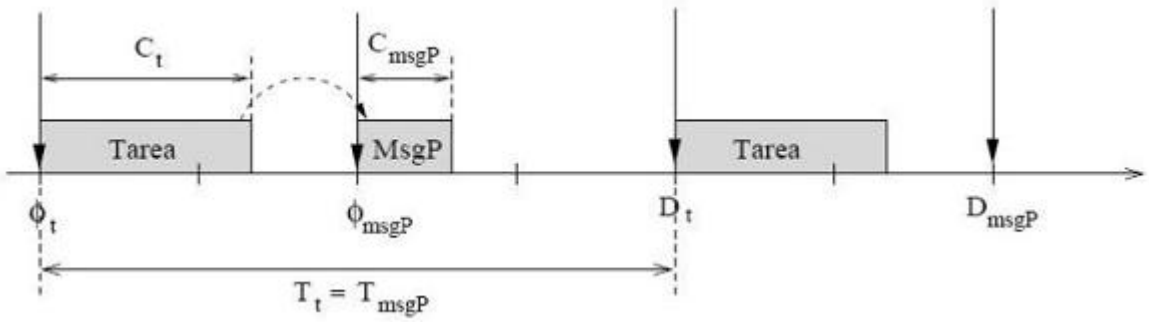

Figura 3.3: Tarea productora

el deadline o plazo de finalización del mensaje, que coincide con el tiempo de respuesta de la aplicación, num_{msg} representa la posición en el mensaje en el nodo; y R_i^m es el tiempo de respuesta del mensaje.

Respecto a la clasificación anterior en la que las tareas dependen de su interacción con el entorno, y basándose en la tesis de *Calha [22]* y la de *Estévez Ayres*, se pueden considerar tres tipos de tareas: consumidoras, productoras, consumidoras/productoras. Las tareas independientes son aquellas que no interactúan con el entorno ni intercambian mensajes.

Tareas productoras. Son las tareas que generan datos. En la figura 3.3 se puede ver que es una tarea que genera un mensaje. Los parámetros previos conocidos son el tiempo de ejecución en el peor caso de la tarea, C_t, y el tiempo de transmisión en el peor caso del mensaje, C_{msgP}. Además, se supone periodos y plazos iguales tanto en la tarea como el mensaje, e iguales entre sí, $T_t = T_{msgP} = D_t = D_{msgP} = T_{DS}$, donde T_{DS} es el periodo del flujo de datos. Por tanto, solo queda por definir el desplazamiento del mensaje, con respecto a la tarea que lo genera:

$$\phi_{msgP} = \phi_t + \left\lceil \frac{C_t}{T_{EC}} \right\rceil T_{EC} \tag{3.8}$$

donde ϕ_t es el desplazamiento de la tarea que genera el mensaje y T_{EC} es la duración del ciclo elemental (EC), que puede definirse como el *slot* de tiempo en el que están dividido tanto el tiempo de ejecución de las tareas como el de transmisión de los mensajes. La fórmula anterior pretende encontrar el instante de tiempo en el que se puede mandar el mensaje generado por la tarea productora, que vendrá definido por el retardo de la tarea, más el número entero de ciclos elementales que necesita la tarea para ejecutarse.

Tareas consumidoras. Es una tarea que necesita datos previos para ejecutarse. Se define como consumidora, entonces, la tarea que necesita la llegada de un mensaje para poder ejecutarse. Como en el caso de las tareas productoras, se impone la condición de periodos iguales, $T_t = T_{msgC} = D_t = T_{DS}$. Así, como se ve en la figura 3.4, debemos calcular el plazo del mensaje consumido y el plazo de la tarea.

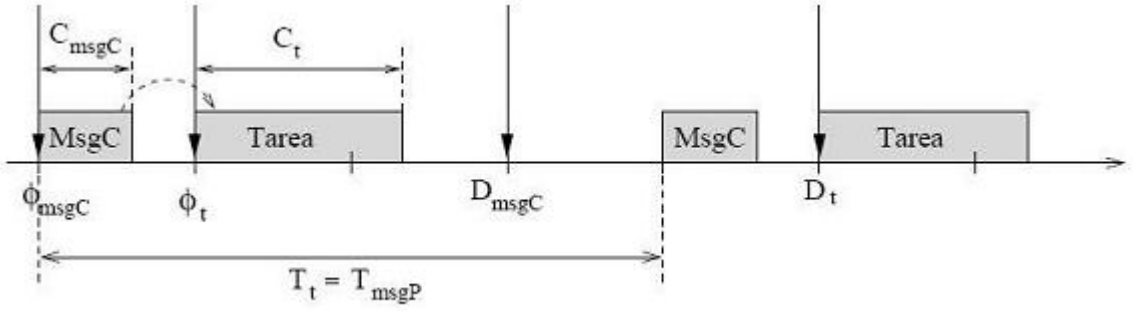

Figura 3.4: Tarea consumidora

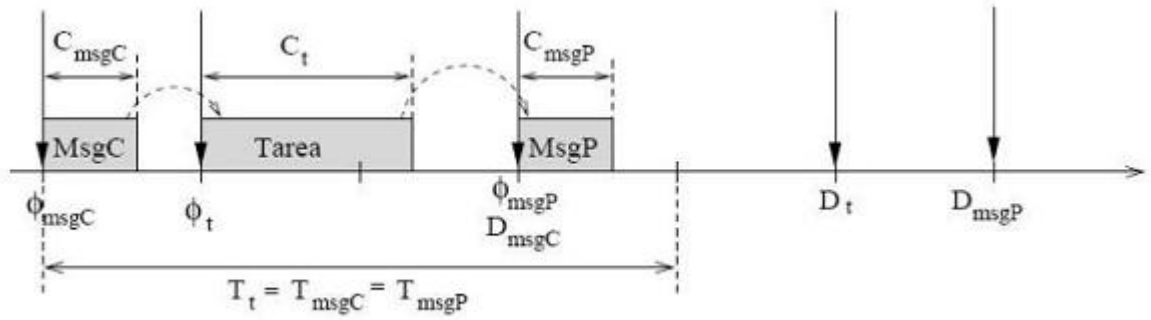

Figura 3.5: Tarea productora-consumidora

El plazo del mensaje consumido es:

$$D_{msgC} = T_{DS} - \left\lceil \frac{C_{msgC}}{T_{EC}} \right\rceil T_{EC} \tag{3.9}$$

y el desplazamiento de la tarea con respecto al mensaje recibido:

$$\phi_t = \phi_{msgC} + \left\lceil \frac{C_{msgC}}{T_{EC}} \right\rceil T_{EC} \tag{3.10}$$

Tareas productoras-consumidoras. Son las tareas que necesitan y generan datos. Es decir, necesita un mensaje para ejecutarse, M_{msgC}, y al finalizar, produce otro mensaje M_{msgP}. En este caso, también se impone la condición de periodos iguales, $T_t = T_{msgC} = T_{msgP} = D_t = T_{DS}$. Por lo que los valores a calcular son el plazo del mensaje consumido, y los desplazamientos de la tarea y y del mensaje producido.

El plazo del mensaje consumido:

$$D_{msgC} = T_{DS} - \left\lceil \frac{C_{msgC}}{T_{EC}} \right\rceil T_{EC} \tag{3.11}$$

el desplazamiento de la tarea con respecto al mensaje recibido:

$$\phi_t = \phi_{msgC} + \left\lceil \frac{C_{msgC}}{T_{EC}} \right\rceil T_{EC} \tag{3.12}$$

y el desplazamiento del mensaje producido con respecto a la tarea que lo genera:

$$\phi_{msgP} = \phi_t + \left\lceil \frac{C_t}{T_{EC}} \right\rceil T_{EC} \tag{3.13}$$

Para finalizar, se deben analizar los puntos de sincronización para una correcta aplicación del modelo utilizado. Los puntos de sincronización son aquéllos donde confluyen varias ramas de un grafo de ejecución. Es decir, el punto donde una tarea consumidora (o productora-consumidora) recibe mensajes de distintas tareas productoras. En consecuencia, determinar los parámetros a calcular de la tarea consumidora dependerá de todas las ramas recibidas, más concretamente, de los valores máximos. En el ejemplo de la figura 3.6 se pueden ver dos puntos de sincronización. El primer punto, P_{S1}, esta en la concurrencia de los servicios B_1 y B_2, que llegan a una tarea consumidora-productora. El segundo punto de sincronización, P_{S2}, surge de la conclusión de los servicios C y B_3, los cuales desembarcan en un único consumidor.

A partir de lo anterior, para una tarea consumidora de n mensajes, $\{M_{msgC}\}_{i=1}^{n}$, y suponiendo que su ejecución empieza al recibir el último de los mensajes que esperaba, las ecuaciones 3.10 y 3.12 quedan modificadas:

$$\phi_t = \max_{\forall i=[1\ldots n]} \left\{ \phi^i_{msgC} + \left\lceil \frac{C^i_{msgC}}{T_{EC}} \right\rceil T_{EC} \right\} \tag{3.14}$$

Decisiones de diseño. La primera decisión que cabe destacar es que, aunque las tareas estén caracterizadas por un desplazamiento u *offset*, este se supone nulo para el cálculo de los tiempos de respuesta de cada tarea; pero por otro lado, sí hay que tenerlo en cuenta para saber el desplazamiento de los mensajes.

Se explicó que para catalogar las tareas atendiendo a su interacción con el entorno, se puede hacer en tres tipos: tareas que necesitan datos, consumidoras; tareas que generan datos, productoras, y tareas que necesitan y generan datos, consumidoras-productoras. En el ejemplo de la figura 3.7 se puede ver una aplicación de lo más sencilla, que consta de tres servicios en cadena, y se aprecia como cada servicio pertenece a cada uno de los tipos:

En primer lugar, la tarea productora que representa al servicio A, será la que, tras su ejecución, creará el mensaje msg_1, que a través de la red llegará la tarea correspondiente al servicio B, la cual es una tarea consumidora-productora, es decir, consumirá el mensaje msg_1, y tras su ejecución producirá el mensaje

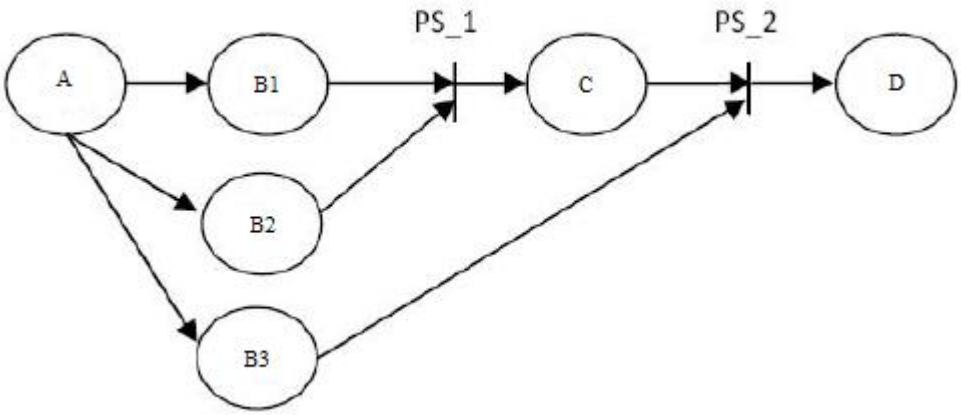

Figura 3.6: Aplicación con dos puntos de sincronización

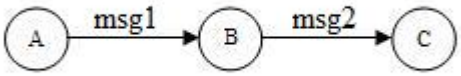

Figura 3.7: Aplicación sencilla

msg_2; este mensaje será consumido por la tarea consumidora que representa al servicio C.

Cada mensaje es considerado independiente y caracterizado como se definió previamente.

Por otro lado, existe otro parámetro relacionado con el T_{EC}, y se trata del T_{LSW}. Mientras que el T_{EC} es la duración del ciclo elemental, el T_{LSW} es la longitud de la ventana de sincronización, es decir, el tiempo máximo de ocupación del ciclo elemental. En este proyecto, LSW tiene un valor del 80 % de la duración del ciclo elemental. Este parámetro surge del paradigma Flexible Time-Triggered (FTT) para sistemas de tiempo real. De lo anterior se desprende que el 20 % restante es usado para el Trigger Message (TM), que como se explicó, es un mensaje de control.

3.4. Decisiones de diseño: resumen y conclusiones

En conclusión, las decisiones de diseño tomadas para los diferentes modelos propuestos crean un sistema gobernado por tiempo. La composición de aplicaciones, orientada a servicios, se realiza seleccionando una serie de estos servicios que son instanciados por una combinación de implementaciones de servicio, las cuales tienen unos parámetros que las caracterizan y afectan al estado del sistema. Dichas implementaciones se comunican mediante mensajes independientes que son gobernados por la red.

3.4.1. Modelo del sistema

- El modelo diseñado consiste en un sistema gobernado por tiempo, suprimiendo uno de los objetivos de la tesis de *Estévez Ayres*: dotar de flexibilidad al sistema en tiempo de ejecución. Este objetivo se convierte en una futura linea de trabajo centrando el trabajo del presente documento en el desarrollo de algoritmos para la composición de aplicaciones.
- Al no tener la flexibilidad en tiempo de ejecución no hay modificaciones ni nuevas implementaciones, debidas, por ejemplo, a sensores que capten datos en tiempo real, lo que implica que el sistema va a ser aplicado es predecible y periódico en todo momento.

3.4.2. Modelo de aplicaciones

- El modelo de aplicaciones consiste en la composición de éstas mediante servicios heterogéneos repartidos por una red de tiempo real.
- Cada aplicación está compuesta por una secuencia de servicios con una configuración específica.
- Los servicios son instanciados por implementaciones, que se comunican entre sí a través de mensajes.
- En definitiva, una aplicación queda definida por una serie de implementaciones de servicio escogidas y los mensajes de red con los que se relacionan.

3.4.3. Modelo de servicios e implementaciones de servicio

- Las implementaciones de servicio son instanciadas en el sistema mediante tareas, caracterizadas como se expuso en la sección anterior. Hay que destacar que aunque las tareas generalmente tengan un *offset* o desplazamiento, inicialmente éste es considerado nulo para el cálculo de sus respectivos tiempos de respuesta.

- Se comentó que para clasificar las tareas atendiendo a su interacción con el entorno, se puede hacer en tres tipos: consumidoras, productoras, consumidoras-productoras. Cada tipo de tarea conlleva una serie de cálculos de sus parámetros, pero, puesto que los desplazamientos de las tareas son considerados nulos inicialmente, el resto de parámetros de dichas tareas se ven simplificados.

- Cada mensaje es considerado independiente y caracterizado por unos parámetros similares a los de las tareas. Además, se consideran como mensajes las interacciones entre procesos del mismo nodo.

- La relación existente entre el T_{EC} y el T_{LSW}. El valor del LSW, es decir, la longitud de la ventana de sincronización, es el 80 % de la duración del ciclo elemental, representado por T_{EC}.

Capítulo 4

Algoritmos de composición

Este capítulo va a desarrollar los diferentes algoritmos de composición de aplicaciones tratados en la tesis de Estévez Ayres, así como las decisiones que se han resuelto para su diseño. Primero se explicará los algoritmos que se han utilizado. Ulteriormente, se presentarán los heurísticos empleados en los algoritmos de mejora. Para finalizar, se comentarán las decisiones de implementación para este proyecto.

4.1. Algoritmos usados

En este último apartado se estudia cómo se realiza la composición de aplicaciones de tiempo real distribuidas basadas en servicios. El principal problema está en la composición de aplicaciones a partir de tareas previamente existentes, que tiene dos problemáticas:

1. Elección de las implementaciones de servicio adecuadas que compondrán la aplicación, en función de unas determinadas restricciones.

2. Asignación de parámetros temporales a las implementaciones de servicio, asegurando que no se pone en peligro la planificación, ni las prestaciones del sistema completo.

Como se puede deducir, ambos subproblemas son co-dependientes, es decir, la solución se ha de presentar de forma global, y no sólo abordando una de las dos variantes del problema existente.

Inicialmente, *Estévez Ayres [9]* propuso un algoritmo de búsqueda exhaustivo, que realiza la composición de aplicaciones buscando una combinación de implementaciones de servicio que minimiza una figura de mérito global, la cual refleja los deseos del usuario y las restricciones impuestas por el propio sistema. Sin embargo, este algoritmo es muy costoso computacionalmente hablando cuando el número de combinaciones es elevado. Por eso, en la tesis de *Estévez Ayres* surgió la idea de realizar un algoritmo heurístico que, en tiempo de ejecución, proponga una combinación subóptima pero lo más similar a la encontrada con el exhaustivo y explorando un número menor de combinaciones. Este algoritmo heurístico se basa en

una figura de mérito relativa que refleja la figura de mérito global; además hace uso de heurísticos que acotan el número de combinaciones a explorar, consiguiendo así, un coste computacional no tan elevado como con el algoritmo exhaustivo.

4.1.1. Algoritmo exhaustivo

Este algoritmo, como se explica al principio de la sección, explora todas las combinaciones posibles de implementaciones de servicio para los servicios solicitados por el usuario de la aplicación. El hecho de comprobar todas las combinaciones existentes y el coste computacional que conlleva hace que no sea eficiente usarse en tiempo de ejecución. Sin embargo, la gran ventaja que acarrea es que encuentra la mejor combinación de implementaciones para la demanda del usuario y las prestaciones del sistema.

El algoritmo no tiene en cuenta el esquema de asignación de prioridades a las tareas, suponiendo que estas vienen impuestas por un desarrollador externo según un esquema de prioridades determinado (por ejemplo basado en bandas de prioridad [25]), mediante el uso de heurísticos [5, 30], o por el propio sistema.

El algoritmo se ejecuta a partir de la petición de un cliente/usuario y el estado actual del sistema. Está basado en el algoritmo usado por *Estévez Ayres [9]* y sigue estos pasos:

1. Previamente al algoritmo de búsqueda propiamente dicho, se descarta de la lista de implementaciones ofrecidas para la composición de la aplicación, todas ellas ordenadas de menor a mayor tiempo de ejecución en el peor caso, aquellas cuyo tiempo de ejecución en el peor caso es mayor que el plazo deseado para la aplicación $C_i > D_{aplic}$, o aquellas que saturen un nodo por el hecho de compartirlo con las tareas previas del sistema, $U_i = \frac{C_i}{T_i} > 1$.

2. Se define el nivel 0 de la aplicación; se trata del primer servicio de la misma.

3. Se enumeran los servicio desde el nivel 0 hasta el nivel N, que será la profundidad máxima de la aplicación.

4. Desde el nivel 0 y recorriendo cada nivel, se elige un camino dentro del grafo de la aplicación:

   a) Se escoge la siguiente implementación de servicio del nivel actual; la primera implementación en caso de ser la primera iteración. Si no existieran más implementaciones en el nivel actual, se evalúa:

      1) Si el nivel actual es el nivel 0, se procede al paso 10.
      2) Se asciende un nivel y se repite el paso 4a.

   b) Si se trata del último nivel, se procede al paso 5.

   c) Se desciende un nivel y se procede al paso 4a.

5. Se verifica que las tareas seleccionadas están en sus nodos correspondientes y se comprueba si la utilización de los nodos físicos sigue siendo posible, es decir, si $U_i = \frac{C_i}{T_i} \leq 1$, teniendo en cuenta ahora las nuevas implementaciones de servicio que componen las aplicaciones.

   a) Si la utilización es mayor que 1, se descarta la combinación y se procede al paso 4a.

6. Se comprueba la planificación de cada nodo físico, asegurando que todas las tareas existentes en el nodo se pueden planificar y cumpliendo que los tiempos de respuesta son menor al plazo requerido.

   a) Si el tiempo de respuesta es mayor al plazo solicitado, se descarta la combinación y se procede al paso 4a.

   b) En caso de ejecutar la parte de código correspondiente a la planificación RUB, se comprueba dicha planificación.

7. Se generan los mensajes necesarios para la comunicación de las tareas seleccionadas y se realiza la planificación de red.

   a) Si la planificación de red no es posible, se descarta la combinación y se procede al paso 4a.

8. Se comprueba si el tiempo de respuesta RTA conseguido con esta combinación es el menor absoluto hasta el momento. En caso afirmativo, guarda la combinación encontrada. Se procede al paso 4a.

9. En caso de ejecutar a parte de código correspondiente al test de planificabilidad RUB, se comprueba que el tiempo de respuesta RUB obtenido es el menor absoluto y se almacena la combinación correspondiente. Además, se calcula el error introducido por la diferencia entre los tiempos RTA y RUB para la misma combinación.

10. Se seleccionan los tiempos obtenidos como los mejores encontrados hasta el momento para el cumplimiento de los requisitos. En caso de no haber encontrado ninguna combinación de implementaciones que cumpla los requisitos, se obtiene una lista vacía.

Una vez calculados los tiempos de respuesta, comprobado que todo es planificable, y asignados los desplazamientos según lo expuesto en el apartado 3.3, donde se supone que el sistema se estructura en ciclos elementales (T_{EC}), es decir, el desplazamiento de cada acción es un múltiplo entero de dicho ciclo, se procede a realizar un informe para la obtención de resultados. Para calcular el desplazamiento en un punto de sincronización, donde llegan varias ramas que se ejecutan a la vez, se aplicará la extensión definida en la ecuación 3.14.

4.1.2. Algoritmo heurístico

Como se adelantó en el apartado anterior, el algoritmo exhaustivo tiene un coste computacional elevado, dependiente del número de combinaciones, el cual se ve incrementado exponencialmente con el número de niveles del grafo de la aplicación. El número de servicios y, en consecuencia, de niveles, no se puede modificar, pero sí se puede acotar el número de combinaciones evaluadas.

Basándose en lo anterior, se presenta un algoritmo heurístico que consiste en una disminución del número de caminos a explorar en el grafo mediante el uso de heurísticos de poda. El algoritmo heurístico aplica una figura de mérito relativa a las implementaciones de cada nivel de la aplicación, creando pequeños grupos o bloques de implementaciones, de forma que sólo se explora un determinado número de caminos del grafo total. La elección del número de implementaciones por nivel a partir del cual se aplica la poda, así como el número de subgrupos y su tamaño, viene dado por el heurístico empleado. Además, cabe destacar que la figura de mérito relativa utilizada ha de estar fuertemente vinculada a la figura de mérito global.

En las siguientes lineas se expondrá el algoritmo heurístico usado en la tesis de *Estévez Ayres*. Inicialmente se procesa una fase preparación, es decir, en el nivel 0 de la aplicación, antes de evaluar ninguna combinación.

1. Previamente, como en el algoritmo exhaustivo, se eliminan aquellas implementaciones cuyo tiempo de ejecución en el peor caso sea mayor que el plazo de la aplicación o cuya instanciación en el nodo físico haga que la utilización de éste supere a la unidad. De nuevo, las implementaciones están ordenadas de menor a mayor tiempo de ejecución en el peor caso.

2. Se realiza una nueva lista de implementaciones cumpliendo una serie de requisitos que aseguran que el número de combinaciones finales va a ser menor que en el caso exhaustivo, es decir, aplicamos un heurístico de poda. Primero se estructuran las implementaciones en bloques de tamaño variable (por defecto dos). La poda se realiza siempre, mediante un número determinado por parámetro de bloques a eliminar, quedando lista de implementaciones de un servicio reducida.

3. Se seleccionan los primeros bloques de cada servicio de la lista, obteniendo una nueva lista mermada de la original, llamada en este proyecto grafo parcial. Se procede a realizar una búsqueda exhaustiva en este grafo parcial. A partir de este paso, el algoritmo actúa igual que para el caso exhaustivo.

4. Desde el nivel 0 y recorriendo cada nivel, se elige un camino dentro del grafo de la aplicación:

   a) Se escoge la siguiente implementación de servicio del nivel actual; la primera implementación en caso de ser la primera iteración. Si no existieran más implementaciones en el nivel actual, se evalúa:

      1) Si el nivel actual es el nivel 0, se procede al paso 6.
      2) Se asciende un nivel y se repite el paso 4a.

   b) Si se trata del último nivel, se procede al paso 5.

   c) Se desciende un nivel y se procede al paso 4a.

5. Se comprueba que el camino elegido es válido para la petición de la aplicación y los tiempos requeridos:

   a) Si el tiempo de respuesta del camino es menor que el tiempo de la combinación almacenada como mejor camino, se guardan las implementaciones del nuevo camino y su tiempo de respuesta como los más óptimos.

   b) Se procede al paso 4a.

6. Se calculan los tiempos para cada posible camino del grafo, independientemente de los demás caminos.

7. Se elige el camino final cumpliendo los requisitos de la aplicación y los caminos paralelos del grafo.

8. Se verifica que las tareas seleccionadas están en sus nodos correspondientes y se comprueba si la utilización de los nodos físicos sigue siendo buena, es decir, si $U_i = \frac{C_i}{T_i} > 1$, teniendo en cuenta ahora las nuevas implementaciones de servicio que componen las aplicaciones:

   a) Si la utilización es adecuada, se omite el paso 9.

   b) Si se satura algún nodo, se procede al paso 9.

9. Se cambia la implementación de servicio del nodo de mayor utilización por la siguiente mejor:

   a) Si se trata del final del bloque actual, se cambia el bloque del servicio correspondiente, se actualiza el grafo parcial, y se procede con el paso 9. Si no quedan más bloques, se da por finalizada la ejecución no consiguiendo cumplir los objetivos. Se procede al paso 4.

   b) Se cambia por la siguiente implementación del bloque actual. Se procede al paso 8.

10. Se comprueba la planificación de cada nodo físico, asegurando que todas las tareas existentes en el nodo se pueden planificar y cumpliendo que los tiempos de respuesta son menor al plazo requerido:

    a) Si algún nodo no es planificable, se procede al paso 11.

    b) Si la planificación es satisfactoria, la combinación de implementaciones de servicio obtenida será la óptima para el estado actual de sistema y los requisitos marcados para la aplicación. Se procede al paso 12.

11. Se cambia la implementación de servicio del nodo que no se puede planificar por la siguiente mejor:

a) Si se trata del final del bloque actual, se cambia el bloque del servicio correspondiente, se actualiza el grafo parcial, y se procede con el paso 11. Si no quedan más bloques, se da por finalizada la ejecución no consiguiendo cumplir los objetivos. Se procede al paso 4.

b) Se cambia por la siguiente implementación del bloque actual. Se procede al paso 10.

12. Se realiza la planificación de red:

a) En caso de que la red sea planificable, se da por válida la aplicación.

b) En caso contrario, se da por finalizada la ejecución no consiguiendo cumplir los objetivos.

De los pasos anteriores hay que hacer notar varias conclusiones. El proceso exhaustivo y el heurístico son procesos similares, con la salvedad de los pasos previos, donde se trata la lista de implementaciones para acondicionarla y procesarla tras la aplicar el heurístico.

También hay diferencia en la forma de explorar el árbol generado, es decir, si con los bloques seleccionados no existe un camino que cumpla los requisitos, se cambia uno de los bloques y se comienza de nuevo una exploración. De esta manera, aseguramos un coste computacional mucho mas bajo, no solo por la poda realizada, sino también por el hecho no explorar todos los caminos posibles inicialmente.

Estas decisiones difieren de la tesis de *Estévez Ayres* en el número de combinaciones a evaluar. En el caso de la tesis, los bloques son seleccionados nivel a nivel, y en caso de que no sea planificable, elige otro bloque del nivel en que se encuentra. Para este proyecto, los bloques se seleccionan al inicio, y se exploran las combinaciones posibles, cambiando de bloque si y sólo si, con los bloques iniciales, el sistema no es planificable.

4.1.2.1. Heurísticos implementados

Estos son los heurísticos desarrollados por *Estévez Ayres*:

Primera combinación válida. Este heurístico plasma el deseo del usuario o cliente de elegir como válida la primera combinación que cumpla los requisitos. El número de implementaciones por bloque es 1, y el número de bloques es el número de implementaciones de ese nivel. Las implementaciones están ordenadas en orden creciente de su valor de figura de mérito relativa, por lo que este heurístico hace una composición, de los mínimos de las figuras de mérito relativas para cada nivel.

Tamaño de bloque fijo. A partir de un tamaño de bloque fijo, este heurístico realiza la poda siempre y cuando el número de implementaciones sea mayor que el tamaño del bloque especificado. Después, se exploran los bloques de tamaño fijo para cada nivel. El número máximo de bloques a evaluar, será la mitad del total más 1.

Dispersión como medida en la realización de la poda. En multitud de ocasiones, la dispersión entre los valores de la figura de mérito relativa no es lo suficientemente grande como para diferenciar cuales son las mejores implementaciones en cada nivel. La introducción en el algoritmo heurístico de una medida de la dispersión mediante la inspección de la relación entre la desviación típica y la media, proporciona una cota para la poda. Si el valor es mayor que dicha cota, y se supera el número de implementaciones en cada nivel, se realiza la poda. Si la condición de superar la cota predeterminada no se cumple, pero se supera el número de implementaciones por nivel, se fuerza la poda, ya que si no, en el peor caso se evaluarían todas las combinaciones.

Tamaño de bloque variable. Puesto que no todos los niveles tienen la misma dispersión entre valores, puede darse el caso de que para un nivel concreto con baja dispersión haya que evaluar un número elevado de implementaciones, mientras que en otros niveles, con mayor dispersión, un número más reducido es suficiente. En este caso, se calcula el tamaño de bloque de tal forma que se deduzca a partir de la cercanía de los valores más bajos de las figuras de mérito a la media.

4.2. Decisiones de diseño de los algoritmos

Para finalizar el capítulo se van a presentar la decisiones tomadas en la elaboración del presente proyecto fin de carrera para el diseño de los algoritmos. Todos los algoritmos han sido escritos en lenguaje de programación *Python*, es decir, un lenguaje de programación orientado a objetos. En consecuencia, la estructura del código se compone de clases y métodos; las tareas, por ejemplo, son instanciaciones de la clase *Tarea*. Un análisis más detallado de todas las clases y los métodos de los que está compuesto el proyecto puede verse en el capítulo 5.

Cabe destacar que la nomenclatura usada en el código del proyecto puede diferir en algunos casos de la usada en el presente documento. Aún así, al igual que en la tesis de *Estévez Ayres*, la misión principal del programa es componer aplicaciones. Las aplicaciones serán evaluadas por diferentes funciones y métodos de análisis dependiendo del estado del sistema en cada momento. Estas funciones y métodos harán un seguimiento de la aplicación desde su preparación previa hasta que el sistema sea planificable tras la composición de la aplicación.

Las aplicaciones están compuestas por servicios, los cuales deben estar previamente definidos a la hora de llamar al sistema, y que forman parte de los parámetros especificados *a priori*. Por tanto, se parte con la premisa inicial de que la entidad encargada de componer las aplicaciones posee toda la información relativa al estado actual del sistema (servicios existentes, sus implementaciones asociadas, la carga de cada nodo físico y la carga total del sistema). Al mismo tiempo, de cada implementación de servicio, se conoce su tiempo de ejecución en el peor caso, pero como lo que se pretende es comprobar la planificabilidad, se debe calcular su tiempo de respuesta. El algoritmo también se encarga, además de seleccionar las implementaciones adecuadas, de establecer el plazo relativo y el desplazamiento temporal (ambos en

relación al T_{EC}) de las tareas que representan a las implementaciones, y de los mensajes intercambiados. En cuanto a otros parámetros temporales, hay que resaltar que existen periodos diferentes, todos de igual valor: el plazo de finalización del nodo; el plazo/periodo del servicio; el plazo/periodo de la red, que se utiliza para la configuración de los mensajes de la red; y el plazo/periodo de la aplicación, que resuelve el el tiempo máximo de respuesta de la aplicación. Sin embargo, el periodo de las tareas previas generadas sí que es diferente.

4.2.1. Decisiones previas

Antes de las decisiones de diseño de los algoritmos, hay que tener en cuenta una serie de decisiones que los condicionan.

Tareas previas. Las tareas previas del sistema son generadas de forma automática, e implementadas en el programa como objetos de la clase *Tarea*. Primero el usuario elige la cantidad de tareas previas de cada nodo y la utilización para ese nodo por parte de dichas tareas, cabe destacar que esto es para el caso de la simulación, pues en un sistema real las tareas previas y la utilización del nodo vienen determinadas por los procesos que ocupen dicho nodo. Posteriormente, el sistema adjudica, con ayuda del algoritmo *UUniFast* (apartado 2.4), una utilización a cada tarea, y haciendo que todas sumen el total previamente elegido. Por otro lado, los periodos son generados mediante una distribución uniforme. Finalmente, se calculan los tiempos de ejecución en el peor caso para cada tarea contando con los datos recién calculados. La asignación de prioridades se hace siguiendo un RMS, concediendo la mayor prioridad a la tarea con mayor frecuencia, es decir, la que tenga menor periodo.

Implementaciones de servicio. Las implementaciones de servicio son generadas también de forma automática, e implementadas en el programa como objetos de la clase *Implementacion_Servicio*. Esta vez son los tiempos de ejecución en el peor caso los que se generan aleatoriamente. Las implementaciones son ordenadas en listas y por servicios. Cada lista se ordena de nuevo de menor a mayor tiempo de ejecución en el peor caso. La prioridad asignada a las implementaciones es siempre la misma, la máxima prioridad, pues se considera que, al tratarse se una aplicación para un sistema de tiempo real, se debe dar prioridad a su composición.

Todas las tareas e implementaciones se guardan en un fichero externo, y posteriormente el programa lee el fichero para simular las condiciones en las que se encuentra cada nodo. En el caso de este proyecto, al tratarse de una simulación, se crean los ficheros necesarios para hacer creer al sistema que está leyendo datos reales de procesos reales.

Una vez leídos los ficheros, tanto las tareas previas como las implementaciones de servicio seleccionadas tras los algoritmos de composición son catalogadas en un diccionario que las ordena por nodos, para un mejor tratamiento a lo largo de la vida del programa.

La comprobación de que cada implementación no conlleve una utilización mayor a la unidad por sí sola y en convivencia con el resto de tareas previas de ese nodo se realiza en la fase previa, como se comentó en la explicación de los algoritmos (apartado 4.1).

Los mensajes, instanciaciones de la clase *Mensaje*, también se generan de forma automática, pero al final del proceso, justo antes de la planificación de red.

4.2.2. Decisiones de diseño del algoritmo exhaustivo

Este algoritmo realiza la comprobación de todas las combinaciones de implementaciones de servicio posibles. Para llevarlo a cabo, se divide la lista de implementaciones ofrecidas en distintos niveles, correspondientes a cada servicio, de manera que nos queda un árbol formado por todas las implementaciones, y cada nivel del árbol se corresponde con un servicio. Después, se explora el árbol evaluando cada camino completo en base a lo analizado en el grafo solicitado por el cliente. Cada camino completo se va evaluando con los requisitos solicitados, y en caso de ser una combinación adecuada, se almacenan los datos necesarios.

Hay que destacar que inicialmente el código se desarrolló paso a paso, por lo que algunas funciones eran específicas para una determinada función. Pero después de terminar el desarrollo de todas las funciones, se cambió todo el esqueleto del programa para trabajar de una forma más eficiente, unificando funciones y eliminando otras que resultaban innecesarias en el nuevo diseño.

Durante la exploración, cada combinación es evaluada para ver si supera cada uno de los requisitos que ha de cumplir para ser válida, como la utilización de los nodos físicos o la planificación del sistema. Si las combinaciones son válidas, se comparar con la mejor de ellas encontrada hasta el momento, y sustituyéndola en caso de que sea mejor. Si una combinación no supera algún requisito, se descarta directamente y se pasa a evaluar otra combinación.

4.2.3. Decisiones de diseño del algoritmo heurístico

El algoritmo heurístico con el uso de heurísticos realiza una comprobación de un menor número de combinaciones posibles para la composición de las aplicaciones. El objetivo es conseguir un menor coste computacional sin perder prestaciones. Esto conlleva a encontrar una solución subóptima, que, teniendo casi las mismas prestaciones y resultados que con el algoritmo exhaustivo, sea mas eficiente computacionalmente hablando.

Primero, se reestructuran las implementaciones existentes agrupándolas en bloques de tamaño fijo. A continuación, y mediante un parámetro, se eliminan una cierta cantidad de bloques.

Así, se selecciona un bloque de cada servicio obteniendo un árbol mucho menor que en el caso de la exploración exhaustiva. Si una combinación no es válida para todas las implementaciones de un bloque, se cambia de bloque. Con esto se pretende conseguir una mayor rapidez a la hora de conseguir encontrar la

solución subóptima, ya que, además de podar el número total de implementaciones, reducimos el número de caminos a explorar.

Capítulo 5

Programa desarrollado

Este capítulo incluye una explicación detallada de cada función, método y porción del código desarrollado para la realización de este proyecto. Esta explicación será exhaustiva, ya que su objetivo es que sirva como complemento para una mayor comprensión del tema que atañe. Posteriormente se explicará el orden en que son ejecutadas dentro del proceso de la composición de las aplicaciones. Algunas funciones están acompañadas de un diagrama de bloques para facilitar la comprensión de las mismas.

5.1. Introducción

A lo largo del capítulo se describen las clases y funciones desarrolladas para la elaboración del proyecto fin de carrera. Primero se detalla el script principal, que es el "main" del programa desarrollado. Después, las clases existentes, y para finalizar, las funciones que se han creado, las cuales están ordenadas según su finalidad.

5.2. Script principal

El script principal son las lineas de código que inicializan variables globales, piden datos de entrada al usuario, generan, si procede, los ficheros externos que se leerán posteriormente, y finalmente ejecuta las principales funciones para la composición de aplicaciones. Las variables globales son:

- **ID**. Es el identificador que se va a usar para que las tareas e implementaciones sean únicos y no se puedan confundir. Se inicializa a 0.
- **servicios_disponibles**. Se trata de una lista de los servicios disponibles. El sistema debe saber que servicios existen previamente a la ejecución del sistema. Por tanto, esta variable debe ser modificada por el usuario con los servicios disponibles.

- **num_implementaciones_por_servicio**. Es el número de implementaciones que se generan de forma automática para cada servicio. Estas serán repartidas de manera aleatoria por cada nodo. Cuánto mayor sea este número, más complejo será el proceso de componer la aplicación de manera exhaustiva, ya que el número de combinaciones crecerá exponencialmente.
- **Cmin**. Es el tiempo de ejecución en el peor caso mínimo que van a tener las implementaciones que se generen.
- **Cmax**. Es el tiempo de ejecución en el peor caso máximo que van a tener las implementaciones que se generen.

A continuación, a través de la consola de ejecución el sistema preguntará al usuario si desea ejecutar el programa en modo artículo o no. En caso afirmativo, el proceso de inicialización de el resto de variables globales será diferente:

- **Tec**. Es la duración del ciclo elemental. Si se ejecuta el programa en modo artículo su valor será de 10000.0 microsegundos; en modo normal será de 100.0 milisegundos.
- **LSW**. Es la longitud de la ventana de sincronización. En modo artículo tiene un valor de 8000.0 microsegundos; en modo normal, de 80.0 milisegundos. Por definición, y como se explicó en el apartado 3.3, su valor es del 80 % del ciclo elemental.
- **D**. Es el plazo de la aplicación, y su valor viene dado por el cliente. En modo artículo tiene un valor de 50000 microsegundos.
- **C_max**. Son los tiempos de ejecución máximos para las implementaciones de los servicios del artículo. Sus valores de inicialización son: [1165.5, 29527.4, 252.0]. Esta variable sólo existe en modo artículo.
- **servicios_disponibles**. Son los servicios existentes en modo artículo. Siempre son los mismos y no se pueden cambiar. Se inicializa con estos servicio: ["A", "B", "C"]. Esta variable solo se inicializa así en modo artículo.

Posteriormente a este proceso, el programa generará los ficheros externos en caso de que se esté simulando. Si se trata de un sistema de tiempo real que existe realmente, deberá omitirse esta porción de código y leer los ficheros de datos reales directamente. Se puede ver un diagrama de flujo general de la aplicación en la figura 5.1.

Finalmente, el sistema ejecuta la búsqueda de las implementaciones que compondrán la aplicación. Dependiendo de lo elegido por el usuario, la búsqueda será exhaustiva para todas las implementaciones existentes o parcial para los heurísticos dados.

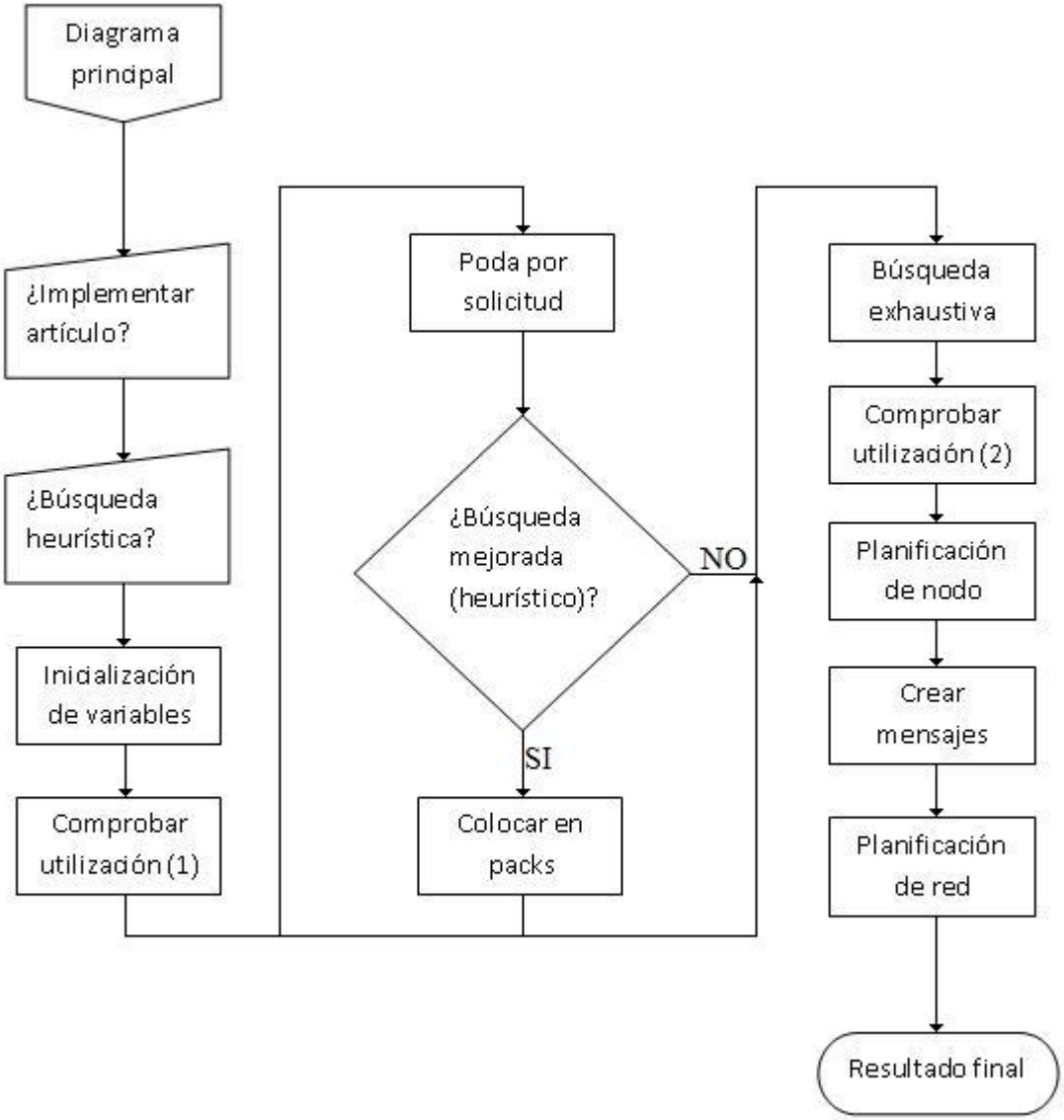

Figura 5.1: Diagrama de flujo general del programa desarrollado

5.3. Clases

Puesto que el programa se desarrolla en *Python*, éste usa un paradigma de programación orientado a objetos. Por tanto, las clases que se definen y se han diseñado para el correcto funcionamiento de este proyecto son *Impelementacion_Servicio*, *Tarea*, y *Mensaje*. Cada una de las clases va acompañada de una figura que resume sus características principales y su constructor para instanciarlo. En dichas figuras el color amarillo corresponde al nombre de la clase, el verde al constructor y el azul a los atributos que posee la clase.

5.3.1. Implementacion_Servicio

La clase *Implementacion_Servicio* hereda de la clase *object*. En la figura 5.2 se puede ver un resumen de la clase. Sus características intrínsecas o atributos son:

- **servicio**. Identificador del servicio que representa la implementación.
- **implementación**. Nombre único de la implementación de servicio. Es el nombre que diferencia una implementación de otra.
- **nombre_nodo**. Nodo de la red en el que esta situada la implementación de servicio.
- **C**. Tiempo de ejecución en el peor caso (WCET: worst case execution time).
- **x**. Desplazamiento de la implementación como tarea. Cuenta desde cero hasta el instante que comienza a ejecutarse.
- **T**. Periodo de la implementación de servicio. Coincide con el valor del plazo de la aplicación.
- **D**. Deadline o plazo de la aplicación. Coincide con el valor del plazo de la aplicación.
- **p**. Prioridad de la implementación. Siempre tendrá el valor de máxima prioridad.
- **Id**. Identificador único que diferencia y hace exclusiva cada implementación de servicio.
- **R**. Tiempo de respuesta de la implementación tras realizar la planificación.
- **Rec**. Tiempo de respuesta de la implementación tras realizar la planificación, representado en EC's.

Para realizar una instanciación de la clase *Implementacion_Servicio* se necesitan una serie de parámetros para su constructor: *def __init__(self, servicio, datos, ind, id)*. Estos parámetros se extraen de un fichero externo y se tratan para que queden escritos en el formato adecuado para su lectura. Con ayuda de estos parámetros, todos los atributos quedan inicializados y el objeto creado. Los parámetros son:

- **servicio**. Es el servicio o funcionalidad que representa la implementación que está creando. Sirve para

- **datos**. Es un diccionario que contiene los datos necesarios de tiempo y nodo para inicializar los atributos.
- **ind**. Es un indice que permite saber a que servicio se está refiriendo, pudiendo seleccionar los datos adecuados del parámetro *datos*: *datos[ind]*. De esta forma, ya que *datos* contiene información de todos los servicios.
- **id**. Es el identificador que dota de exclusividad a la implementación. Da valor al atributo *Id*.

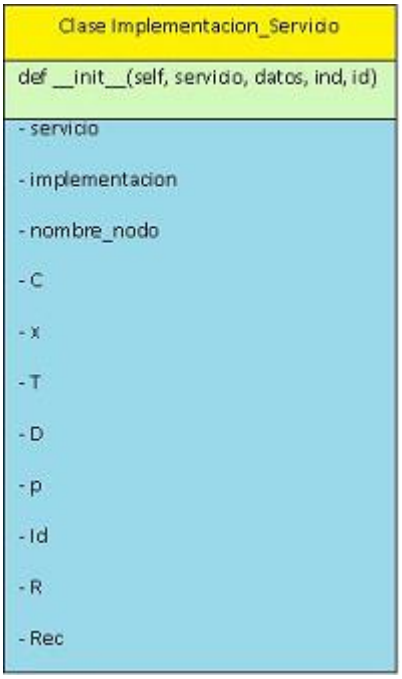

Figura 5.2: Clase Implementación_Servicio

Es destacable que los valores de los tiempos de respuesta quedan incializados al valor del tiempo de ejecución en el peor caso, hasta que se haga el cálculo de su verdadero valor en la planificación.

5.3.2. Mensaje

Los mensajes de red son objetos de la clase *Mensaje*, la cual también hereda de *object*, véase figura 5.3. Los atributos se los que se compone son:

- **servicio_origen**. Es el servicio que produce el mensaje.
- **servicio_destino**. Es el servicio destinatario del mensaje, será el que lo procesará.

- **C**. Tiempo de transmisión en el peor caso (WCTT: worst case transmission time)
- **x**. Desplazamiento del mensaje. Cuenta desde cero hasta el instante que comienza a ejecutarse.
- **T**. Periodo del mensaje.
- **D**. Deadline (plazo) del mensaje.
- **Id**. Identificador único que hace exclusivo cada mensaje.
- **R**. Tiempo de respuesta del mensaje tras realizar la planificación.
- **Rec**. Tiempo de respuesta del mensaje tras realizar la planificación, representado en EC's.

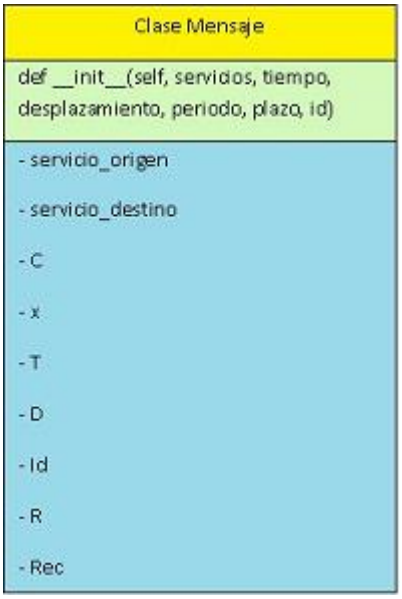

Figura 5.3: Clase Mensaje

Para inicializar un objeto mensaje y sus atributos usamos, como en otros casos, su constructor: *def __init__(self, servicios, tiempo, desplazamiento, periodo, plazo, id).* A la vista de los parámetros que se piden en el constructor, se desprende que sirven para inicializar cada uno de los atributos. el parámetro *servicio* es una tupla que contiene la información necesaria para los atributos *servicio_origen* y *servicio_destino*. De nuevo, los tiempos de respuesta son inicializados al valor de tiempo de transmisión en el peor caso, a espera del cálculo del verdadero valor para dicho atributo.

5.3.3. Tarea

La clase *Tarea* esta diseñada para representar las tareas previas existentes en el sistema. Por tanto, también hereda de la clase *object*, y las tareas previas serán objetos de esta clase. Los atributos de esta clase no difieren mucho de los de la clase Implementacion_Servicio:

- **C**. Tiempo de ejecución en el peor caso (WCET: worst case execution time).
- **x**. Desplazamiento de la tarea. Cuenta desde cero hasta el instante que comienza a ejecutarse.
- **T**. Periodo de la tarea.
- **D**. Plazo de la tarea.
- **p**. Prioridad asignada a la tarea.
- **Id**. Identificador único que hace exclusiva cada instanciación de la clase.
- **R**. Tiempo de respuesta de la tarea tras realizar la planificación.
- **Rec**. Tiempo de respuesta de la tarea tras realizar la planificación, representado en EC's.

Los parámetros de entrada de la clase *Tarea* sirven para inicializar cada uno de los atributos (véase figura 5.4), y todo se realiza a través de su constructor: *def __init__(self, tiempo, desplazamiento, periodo, plazo, prioridad, id)*. Nótese que los tiempos de respuesta se inicializan al valor del parámetro *tiempo*, hasta que posteriormente se haga la planificación y se calcule su verdadero valor.

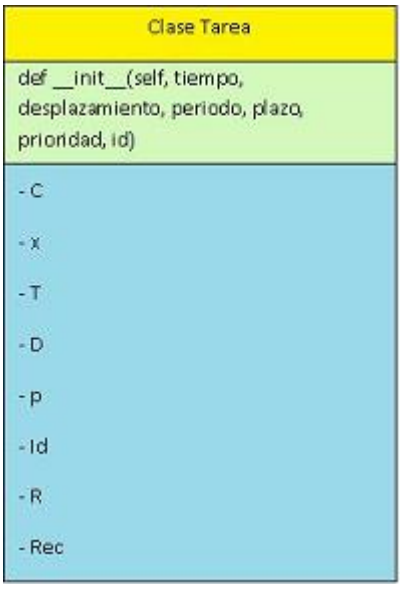

Figura 5.4: Clase Tarea

5.4. Funciones

A continuación se detallan todos las funciones de las que se compone el programa, explicando cada parámetro de entrada, cada parámetro de salida, así como el procedimiento que sigue para su realizar su funcionalidad.

5.4.1. Funciones relativas a la estructuración de las implementaciones

5.4.1.1 limpiar_nodos

El principal cometido de la función *limpiar_nodos* es eliminar del diccionario tareas_por_nodos todas las implementaciones de servicio y dejar solo las tareas previas. Para ello, recorre el diccionario que contiene las tareas e implementaciones ordenadas por nodos comprobando una a una y eliminando únicamente las implementaciones de servicio.

def limpiar_nodos(tareas_por_nodos)

...

return tareas_por_nodos

Su único parámetro de entrada y de salida es:

- **tareas_por_nodos**. Es un diccionario, en el cual, cada clave del diccionario representa un nodo, y contiene una lista con todas las tareas previas e implementaciones de servicio existentes en ese nodo.

5.4.1.2 leer

La función *leer* se encarga de interpretar los ficheros externos de tareas previas, implementaciones, y grafo solicitado por el cliente para guardarlo como variables que el programa pueda entender. Los ficheros pueden ser generados de manera automática para la simulación, pueden ser datos reales de un sistema de tiempo real. Por tanto, esta definida por:

def leer()

...

return implementaciones, grapho, tareas

No existen parámetros de entrada, y los parámetros de salida no son más que las implementaciones de servicio existentes para la composición de servicios, el grafo solicitado por el cliente, y las tareas alojadas en los nodos de manera previa a la ejecución del sistema. El funcionamiento es simple: leer el determinado fichero, y adaptarlo a una formato compatible con el lenguaje de programación.

5.4.1.3 list_implementation

Esta función es la encargada de generar los objetos Implementacion_Servicio a partir de los datos de entrada. Por tanto, *list_implementation* recibe información en crudo de las implementaciones y devuelve las listas de objetos de implementaciones de servicio. Su definición es:

def list_implementation(servicio, data)

...

return imp

Los parámetros de entrada son:

- **servicio**. Es el servicio en cuestión que se va a tratar en la ejecución de la función.
- **data**. Son los datos en crudo de todas las implementaciones existentes en el sistema.

Y el único parámetro de salida es *imp*, que es la lista de objetos Implementacion_Servicio del servicio que se va a tratar en la ejecución de la función. El funcionamiento consiste en leer la parte de *data* correspondiente a la funcionalidad dada en la variable *servicio*, e instanciar las implementaciones de servicio pasándole los datos necesarios a su constructor.

5.4.1.4 ordenar_implementaciones

La función *ordenar_implementaciones* recoloca las listas de implementaciones de servicio de manera que queden ordenadas de menor a mayor tiempo de ejecución en el peor caso. Esta definida así:

def ordenar_implementaciones(grafo_ofrecido)

...

return grafo_ofrecido_ordenado

Su único parámetro de entrada es el grafo de implementaciones ofrecidas sin orden alguno, pero colocadas por servicio. Y el único parámetro de salida es el mismo grafo que el de entrada, pero además de estar colocadas por servicios, las implementaciones quedan ordenadas por su tiempo de ejecución. Para conseguir su objetivo la función guarda los tiempos de cada una de las implementaciones en una lista, ordena los tiempos de dicha lista de menor a mayor, y vuelve a crear un nuevo diccionario siguiendo el nuevo orden dado por los tiempos.

5.4.1.5 ordenar_datos

La función *ordenar_datos* tiene por objetivo crear los datos en crudo de las implementaciones, a partir de los datos leídos del fichero. Es decir, es el paso intermedio de la conversión de los datos del fichero en datos en crudo para la creación de las instancias de las implementaciones de servicio. Su definición es:

def ordenar_datos(implementaciones)

...

return datos

Su parámetro de entrada es la variable que contiene los datos leídos del fichero, y su único parámetro de salida son los datos en crudo que sirven para crear las instanciaciones de las implementaciones de servicio. Su funcionamiento consiste en solamente en leer y adaptar los mencionados datos.

5.4.1.6 ordenar_listas

La función *ordenar_listas* se encarga de recibir los datos en crudo de las implementaciones y devolver las listas de objetos Implementacion_Servicio. Para ello, va tratanto servicio a servicio y con ayuda de la función *list_implementation* va creando los objetos para devolverlos en una lista. Está definida por:

def ordenar_listas(datos)

...

return listas

El parámetro de entrada son los datos en crudo adaptados del fichero por la función *ordenar_datos* y el parámetro de salida son las listas que incluyen ya las instancias de las implementaciones de servicio.

5.4.1.7 distribucion_uniforme

Como su nombre indica, la función *distribucion_uniforme* crea números siguiendo una distribución uniforme a partir de dos umbrales dados. Está definida por:

def distribucion_uniforme(n, Tmin, Tmax)

...

return vect_T

Los parámetros de entrada son: *n*, que es el número de elementos que debe generar, y *Tmin* y *Tmax*, los umbrales para definir la distribución uniforme. Su uso viene dado por la necesidad de generar periodos aleatorios para la creación de las tareas previas simuladas, y de ahí viene el nombre de su parámetro de salida, *vect_T*, que es un vector o lista que contiene los periodos para las tareas previas.

5.4.1.8 UUniFast

La función *UUniFast* se encarga de crear valores de utilización para un cantidad de elementos dada y una utilización media para un nodo concreto. Para llevar a cabo su misión, ejecuta el algoritmo UUniFast (apartado 2.4), y almacena los valores en una lista para su tratamiento posterior. Esta definida así:

def UUniFast(n, U_media)

...

return vect_U

Los parámetros de entrada son: *n*, el número de elementos, es decir, el número de tareas previas alojadas en el nodo; y *U_media*, que el valor de utilización total existente en el nodo para repartir entre todas las

tareas. El parámetro de salida, *vect_U*, es un lista que contiene los valores de utilización adjudicados a cada tarea.

5.4.1.9 calculo_tareas

La función *calculo_tareas* hace una simple multiplicación entre los periodos y las utilizaciones de las tareas previas para obtener los valores de los tiempos de ejecución en el peor caso. Por tanto, viene definida como:

def calculo_tareas(vect_U, vect_T)

...

return vect_C

Donde los parámetros de entrada son los vectores o listas de valores de periodos y utilizaciones, y el parámetro de salida es *vect_C*, una nueva lista que contiene tantos tiempos de ejecución el peor caso como tareas vaya a existir en un nodo dado.

5.4.1.10 aniadir_tareas

La función *aniadir_tareas* es la encargada de colocar las implementaciones de servicio elegidas para la aplicación en el diccionario de tareas que están colocadas por nodos. Su definición:

def aniadir_tareas(nodos, grafo_elegido)

...

return nodos

Sus parámetros de entrada son el diccionario de tareas clasificadas por nodos y la combinación válida de implementaciones de servicio. El parámetro de salida es el mismo diccionario de tareas clasificadas por nodos, pero con las implementaciones añadidas en su correspondiente nodo. Por tanto, su misión es recorrer la lista de implementaciones de servicio, y colocarlas en el diccionario en su nodo correspondiente.

5.4.1.11 crear_tareas

El cometido de la función *crear_tareas* es instanciar las tareas previas del sistema como objetos de la clase Tarea. Esto lo realiza a partir de los datos en crudo leídos del fichero externo, independientemente de si éste ha sido generado para la simulación o son tareas reales escritas en un fichero. Esta definido por:

def crear_tareas(tareas)

...

return nodos

El parámetro de entrada, *tareas*, son los datos en crudo que contiene información de las tareas previas, mientras que el parámetro de salida, *nodos*, es un diccionario que contiene las tareas, ya instanciadas, organizadas por nodos.

5.4.1.12 escribir_tareas

La función *escribir_tareas* tiene como objetivo escribir la información generada previamente de las tareas previas en un diccionario. Su estructura es ésta:

def escribir_tareas(periodos, C_s, letra):

...

return lista_tareas_previas_nodo

Los datos de entrada son los necesarios para poder instanciar una tarea: periodos, tiempos de ejecución en el peor caso, y el nodo al que pertenecen. De esta forma se consigue reunir la información generada para simular las tareas previas de un nodo. La salida son los datos en crudo, que posteriormente se escribirán en un fichero.

5.4.1.13 escribir_fichero

Esta función no hace otra cosa que generar el fichero correspondiente a las tareas previas que le pasa como parámetro. Esta información viene dada en crudo, por lo que el programa deberá tratarla para su correcto funcionamiento en la composición de aplicaciones. Cabe destacar que *escribir_fichero* no tiene parámetros de salida:

def escribir_fichero(file_data)

...

5.4.1.14 escribir_implementaciones

La función *escribir_implementaciones* crea los datos en crudo de información relacionada con las implementaciones de servicio, que van a ser escritos en un fichero externo. Viene definida así:

def escribir_implementaciones(C_s_implementaciones, letra, max)

...

return lista_implementaciones

Los parámetros de entrada son: *C_s_implementaciones*, que son los tiempos de ejecución en el peor caso de cada implementación; *letra*, que es el servicio al que corresponden los tiempos; y *max*, que es el número máximo de nodos existentes en el sistema. El único parámetro de salida es un diccionario que contiene toda la información básica, preparada para ser escrita en un fichero de texto externo.

5.4.1.15 escribir_implementaciones_articulo

La función *escribir_implementaciones_articulo* crea los datos en crudo de información relacionada con las implementaciones de servicio del caso particular del artículo, que van a ser escritos en un fichero externo. Viene definida así:

def escribir_implementaciones_articulo(C_s_articulo, letra, max):

...

return implementaciones_articulo

Los parámetros de entrada son: *C_s_*articulo, que son los tiempos de ejecución en el peor caso de cada implementación del artículo; *letra*, que es el servicio al que corresponden los tiempos; y *max*, que es el número máximo de nodos existentes en el sistema. El único parámetro de salida es un diccionario que contiene toda la información básica, preparada para ser escrita en un fichero de texto externo.

5.4.1.16 elimina_implementaciones

El funcionamiento de *elimina_implementaciones* es sencillo, consiste en eliminar de la lista de implementaciones de servicio existentes en el sistema aquéllas que tengan un tiempo de ejecución en el peor caso mayor que el plazo de la aplicación, ya que se consideran tareas no usables para la composición de la aplicación que se está formando y por eso se descartan. Su único parámetro de salida y entrada es el diccionario de implementaciones ordenadas por servicios:

def elimina_implementaciones(grafo_ofrecido)

...

return grafo_ofrecido

5.4.1.17 colocar_en_packs

La función *colocar_en_packs* reestructura el diccionario de implementaciones de servicio ofrecidas ordenando las mencionadas implementaciones en tuplas de varios elementos. El número de elementos viene dado por *implementaciones_por_pack*, una variable local que por defecto es 2. Por tanto, su parámetro de entrada es el diccionario de implementaciones colocadas por servicios y la salida es un diccionario, idéntico al anterior, salvo que las implementaciones están colocadas por parejas:

def colocar_en_packs(grafo_ordenado)

...

return grafo_ordenado_en_packs

5.4.2. Funciones relativas a la exploración y obtención de combinaciones

5.4.2.1 busqueda_exhaustiva

La función *busqueda_exhaustiva* tiene como objetivo realizar una exploración de un grafo dado para encontrar el mejor camino posible de entre todas las posibles combinaciones. Es una función autorecurrente que se organiza por niveles, donde cada nivel del árbol se corresponde con un servicio. Cabe destacar que esta función no devuelve el mejor camino con todas las comprobaciones de utilización y planificación realizadas, si no que solo tiene en cuenta los tiempos, y es usada en ambos algoritmos de búsqueda: exhaustivo y heurístico. Queda definida así:

def busqueda_exhaustiva(grafo_ofrecido, R, best=[], path=[], nivel=0)
...
return best, R, nivel-1
Los parámetros de entrada son:

- **grafo_ofrecido**. Es un diccionario con las implementaciones ofrecidas. Es el árbol que va a explorar rama por rama para encontrar una solución óptima.
- **R**. Es el tiempo de respuesta conseguido por el camino. Como solo se hace por comprobación de tiempos, es la suma de todos los tiempos de ejecución en el peor caso de las implementaciones elegidas al final de la exploración. Inicialmente, su valor es el plazo de la aplicación.
- **best**. Es una lista, inicialmente vacía, que almacena el mejor camino explorado hasta el momento. Al final de toda la búsqueda, será la combinación elegida como válida para la aplicación a falta de comprobar utilización en los nodos y planificar el sistema.
- **path**. Se trata de una lista que guarda los nombres de los servicios que se van recorriendo por cada rama del grafo general. Sirve para que el programa se oriente por el árbol. Si la exploración llega un nodo-hoja del árbol, el path es susceptible de convertirse en un camino válido.
- **nivel**. Es una variable numérica que nos indica en que nivel del árbol se encuentra la exploración. Inicialmente se encuentra en el nivel 0, en la raíz del árbol. Cada vez que la función es ejecutada, el nivel se ve incrementado en 1, y cada vez que la función retorna hacia atrás en la rama, el nivel se decrementa en 1.

Y los parámetros de salida son:

- **best**. Es la mejor combinación conseguida hasta el momento. En caso de estar en el nivel 0 de la aplicación, se está devolviendo una combinación válida a falta de, como se mencionó antes, ciertas comprobaciones.
- **R**. Es el tiempo de respuesta conseguido por la combinación de implementaciones guardada en *best*. Este tiempo sufrirá modificaciones en tratamientos posteriores.
- **nivel-1**. Sirve para saber en que nivel del árbol nos encontramos. Aparece restando -1 ya que al utilizar el return estamos volviendo atrás en la rama y por tanto, bajando un nivel. En el caso de encontrarnos en el nivel 0, el número es innecesario como se mostró en el script principal.

5.4.2.2 comprobar_RTA

La función *comprobar_RTA* esta diseñada para analizar cada una de las combinaciones encontradas por la búsqueda exhaustiva y verificar si cumplen las condiciones impuestas por el cliente. Esta definida de siguiente forma:

def comprobar_RTA(path_elegido, lista_caminos)
...
return False/True, Tiempo_RTA
Los parámetros de entrada son:

- **path_elegido**. Es el camino elegido previamente en la exploración del árbol, el cual va a ser comprobado para ver si es una combinación válida.
- **lista_caminos**. Son los diferentes caminos existentes en el grafo solicitado por el cliente. Se necesitan para analizar cada camino por separado.

Y los parámetros de salida son:

- **True/False**. Es un *booleano* que es *True* en caso de que el tiempo de respuesta de la combinación elegida sea menor que el plazo de la aplicación y es *False* en caso contrario.
- **Tiempo_RTA**. Es el tiempo de respuesta de la combinación analizada según el método clásico.

La función *comprobar_RTA* hace todas las comprobaciones necesarias para saber si una combinación es válida o no. Para ello, primero comprueba que la combinación elegida satura algún nodo comprobando su utilización. En segundo lugar, calcula los tiempos de respuesta de las tareas para dicha combinación y si es planificable. En caso afirmativo, calcula los tiempos de respuesta del sistema completo con los tiempos de los mensajes incluidos y si es planificable la red. Finalmente, si todas las comprobaciones son correctas, almacena los datos necesarios en caso de ser una combinación mejor que la almacenada anteriormente.

5.4.2.3 comprobar_RUB

La función *comprobar_RUB* esta diseñada para analizar cada una de las combinaciones encontradas por la búsqueda exhaustiva y verificar si cumplen las condiciones impuestas por el cliente. Esta definida de siguiente forma:

def comprobar_RUB(path_elegido, lista_caminos)
...
return False/True, Tiempo_RUB
Los parámetros de entrada son:

- **path_elegido**. Es el camino elegido previamente en la exploración del árbol, el cual va a ser comprobado para ver si es una combinación válida.
- **lista_caminos**. Son los diferentes caminos existentes en el grafo solicitado por el cliente. Se necesitan para analizar cada camino por separado.

Y los parámetros de salida son:

- **True/False**. Es un *booleano* que es *True* en caso de que el tiempo de respuesta de la combinación elegida sea menor que el plazo de la aplicación y es *False* en caso contrario.
- **Tiempo_RUB**. Es el tiempo de respuesta de la combinación analizada según el método clásico.

Como se puede observar, el diseño de *comprobar_RUB* es igual al de *comprobar_RTA*. La principal diferencia, además de referirse a tiempos y combinaciones para el test de planificabilidad RUB, es que la llamada a la planificación es distinta. Mientras que *comprobar_RTA* llama a la función *planificacion_nodo*, la función *comprobar_RUB* llama a la función *planificacion_rub*.

5.4.2.4 guardar

La función guardar tiene por objetivo almacenar la mejor combinación de implementaciones encontrada hasta el momento en la exploración que se este ejecutando en ese momento. Para ello, recibe los datos candidatos a ser la mejor combinación, y con los resultados de la función comprobar, almacena o no dichos datos en las variables del mejor combinación y mejor tiempo. Queda definida así:

def guardar(caso, path, mejor, R)

...

return mejor, R

Los parámetros de entrada son:

- **caso**. Dependiendo de si caso tiene valor 1 o 2, se procederá con tests planificabilidad RTA o RUB, respectivamente.
- **path**. El camino recibido como posible mejor combinación de implementaciones.
- **mejor**. La mejor combinación existente hasta el momento en la exploración del árbol.
- **R**. El tiempo de respuesta de la mejor combinación existente hasta el momento.

Y los parámetros de salida son:

- **mejor**. Es la mejor combinación de implementación se servicio que se tiene hasta el momento, haya sido sustituido por una nueva o no.
- **R**. Es el tiempo de respuesta de la mejor combinación conseguida hasta el momento.

El funcionamiento de la función es básico: A partir de los datos de entrada, calcula los posibles caminos del grafo solicitado por el cliente y llama a la función *comprobar_RTA* o *comprobar_RUB* administrándole los datos necesarios. En base a la respuesta de *comprobar_RTA/comprobar_RUB*, toma la decisión de guardar el camino y su correspondiente tiempo en sendas variables y devuelve dichas variables.

5.4.2.5 busqueda_mejorada

La misión principal de la función *busqueda_mejorada* es aplicar el heurístico de poda al conjunto de implementaciones de servicio y encontrar una combinación de implementaciones adecuada a los requisitos del cliente a partir de un árbol de exploración menor que el completo. La función queda definida de esta forma:

def busqueda_mejorada(grafo_ofrecido_ordenado, tareas_por_nodos, grapho)

...

return grafo_final, Resp_App

Los parámetros de entrada son:

- **grafo_ofrecido_ordenado**. Es el diccionario que contiene todas las implementaciones de servicio existentes en el sistema ordenadas por tiempo de ejecución en el peor caso y catalogadas por servicios.
- **tareas_por_nodos**. Es el diccionario que contiene las tareas e implementaciones alojadas en cada nodo.
- **grapho**. Es la grafo solicitado por el cliente para formar su aplicación.

Y los parámetros de salida son:

- **grafo_final**. Es la combinación de implementaciones encontrada tras la exploración en un árbol podado con heurísticos.
- **Resp_App**. Es el tiempo de respuesta de la aplicación dado por el programa para el caso de realizar heurísticos de poda.

Su funcionamiento consiste en reestructurar las implementaciones colocándolas en tuplas con ayuda de la función *colocar_en_packs*, posteriormente realizar la poda del grafo completo y seleccionar algunas de esas tuplas para obtener un árbol de exploración más pequeño. Finalmente, se hace una llamada a *proceso_completo*, que es la función que obtiene la combinación y tiempo de respuesta para la aplicación exigida.

5.4.2.6 encontrar_extremos

La función *encontrar_extremos* explora el grafo solicitado por el cliente para conocer los servicios de inicio y fin, es decir, el primer servicio que solicita el cliente, y con el que concluye el grafo. La función queda definida por:

def encontrar_extremos(grafo_solicitado)

...

return extremos

El único parámetro de entrada es *grafo_solicitado*, que no es más que la aplicación requerida por el cliente, y el único parámetro de salida es *extremos*, una lista que contiene el nombre de los servicios de inicio y fin. El funcionamiento es sencillo: con ayuda de bucles *for* anidados se analiza todo el grafo, obteniendo una lista de posibles servicios origen y otra de posibles servicios destino. Los servicios que sean los de inicio y fin se consiguen aplicando un condicional a cada una de las listas.

5.4.2.7 find_all_paths

Esta función, *find_all_paths*, encuentra todos los caminos posibles en un grafo con ramas en paralelo. Dado un grafo, localiza cada camino desde el inicio hasta el final del grafo y lo almacena en una lista que contiene todos los caminos de servicios del grafo. La función es autorecurrente. Esta definida como:

def find_all_paths(grapho, start, end, path=[])

...

return paths

Los parámetros de entrada son: el grafo sobre el cual hace la búsqueda de todos los caminos, *grapho*; dos variables internas, *start* y *end*, que sirven de apoyo para encontrar el camino eficientemente; y *path*, una variable interna que va almacenando el camino parcial hasta que el camino completo es encontrado. El único parámetro de salida, *paths*, es la lista que contiene cada uno de los caminos encontrados en el grafo.

5.4.3. Funciones relativas al cumplimiento de los requisitos de la aplicación

5.4.3.1 utilizacion

La función *utilizacion*, como su nombre indica, es la encargada de hacer las comprobaciones de utilización de los nodos, y en caso de existir saturación en algún nodo, tomar medidas al respecto. Queda definida de la siguiente manera:

utilizacion(caso, nodos, grafo_ofrecido_ordenado)

...

1. ***return grafo_ofrecido_ordenado***
2. ***return True/False***

Los parámetros de entrada son:

- **caso**. Es una variable numérica que nos dice qué tipo de utilización se va a analizar, actúa a modo de switch.
- **nodos**. Es un diccionario que contiene las tareas y/o implementaciones ordenadas por nodos.

- **grafo_ofrecido_ordenado**. Es el conjunto total de implementaciones ordenadas por su tiempo de ejecución en el peor caso.

Los parámetros de salida dependen de qué porción de código se ejecuta:

1. Si caso = 1: devuelve el conjunto total de implementaciones ordenadas después de haber comprobado si cada una de las implementaciones, junto a las tareas alojadas en su correspondiente nodo, satura dicho nodo.

2. Si caso = 2: devuelve una variable booleana que nos indica con *True* si las comprobaciones de utilización son satisfactorias y *False* en caso contrario.

El funcionamiento de esta función, véase figura 5.5, depende de nuevo de cuál sea el caso que se está ejecutando:

1. Caso = 1. Se comprueba que las tareas previas no superan la utilización del nodo. Si es así, con ayuda de la función *utilizacion_priori*, comprobamos las implementaciones de manera individual.

2. Caso = 2. Se comprueba la utilización de cada nodo una vez conocidas qué implementaciones son las elegidas para componer la aplicación. Se realiza con ayuda de la función *utilizacion_posteriori*.

5.4.3.2 utilizacion_posteriori

Como se comentó en *utilizacion* (sección 5.4.3), *utilizacion_posteriori* se ocupa de comprobar la utilización de cada uno de los nodos con todas las tareas alojadas en él, tanto previas como implementaciones de servicio de la combinación elegida para la aplicación. En caso de que uno de los nodos supere la unidad, la función determina un False que hace que la combinación no sea válida. Queda definida de la siguiente manera:

def utilizacion_posteriori(nodos)

...

return True/False

Como se comentó en el apartado 5.4.3, el parámetro de entrada *nodos*. Y su parámetros de salida es una variable booleana, que indica si la utilización supera o no la unidad.

5.4.3.3 utilizacion_priori

Como se explicó en el apartado 5.4.3, *utilizacion_priori* comprueba la utilización de cada nodo con las tareas previas existentes en dicho nodo y cada implementación de servicio de manera individual. En caso de alguna de las implementaciones sature el nodo, es descartada de la lista de candidatas a ser elegida como parte de la aplicación. De esto último se desprende que este proceso se realiza de manera previa a la búsqueda exhaustiva. Queda definida así:

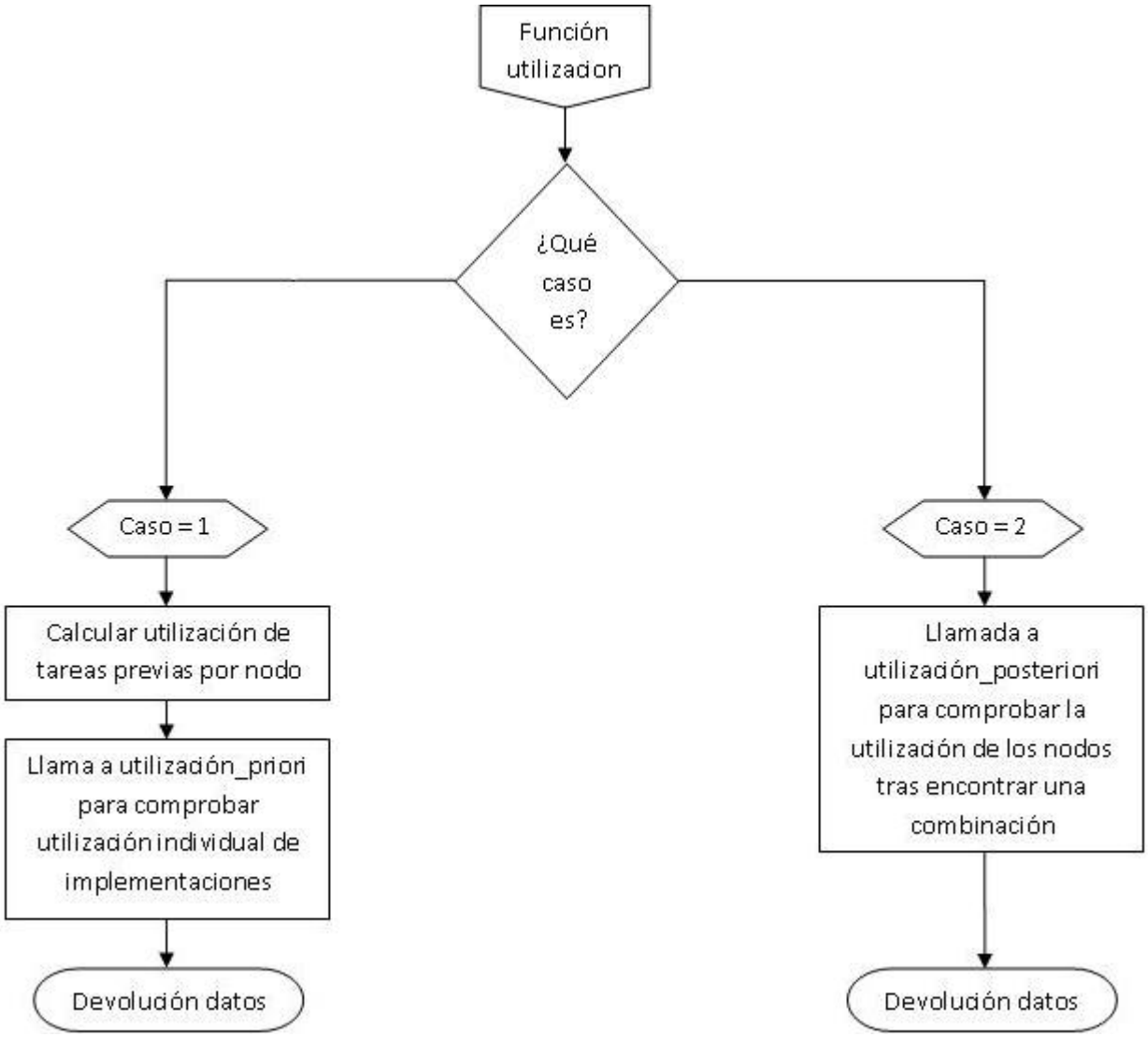

Figura 5.5: Diagrama de flujo de la función utilización

def utilizacion_priori(nodos, grafo_ofrecido_ordenado)

...

return grafo_ofrecido_ordenado

Los parámetros de entrada con el diccionario de tareas organizadas por nodos, y el diccionario que contiene todas las implementaciones existentes organizadas por servicio. El único parámetro de salida es el mismo diccionario de implementaciones, del cual han sido eliminadas las implementaciones que no pueden usarse para la composición de la aplicación que atañe en ese momento.

5.4.3.4 planificacion_nodo

La función *planificacion_nodo* tiene como objetivo comprobar si un nodo es planificable dado un conjunto de tareas e implementaciones. Para ello, se calcula el tiempo de respuesta para cada una de las tareas existentes en el nodo a partir de la fórmula dada por el algoritmo de planificación RTA, donde se tiene en cuenta que el tiempo de respuesta de una tarea solo se ve afectado por otras tareas de igual o mayor prioridad. La función viene definida como:

def planificacion_nodo(tareas_nodo)

...

return True/False

Donde tareas_nodo es el parámetro de entrada y es una lista de las tareas e implementaciones dadas para un nodo en particular. Y el parámetro de salida es una variable booleana que informa si la planificación ha sido satisfactoria o no, para proceder de la forma adecuada.

5.4.3.5 planificacion_rub

La función *planificacion_rub* sigue el algoritmo de planificación detallado en el apartado 2.2.1.1, es decir, hace una planificación que advierte de si el sistema es planificable de extremo a extremo o no. La función viene definida por:

def planificacion_rub(tareas_nodo)

...

return True/False

El único parámetro de entrada en el diccionario de tareas catalogadas por nodo, del cual la función va a hacer uso. El parámetro de salida existente es una variable booleana es *True* en caso de que la planificación extremo a extremo sea válida y *False* en caso contrario.

5.4.3.6 crear_mensajes

La función *crear_mensajes* tiene como objetivo crear las instanciaciones de los mensajes para el sistema, es decir, crea los objetos de la clase *Mensaje* y los inicializa con unos valores predeterminados,

incluyendo el servicio que genera el mensaje y a cuál va destinado el mismo. Es una función autorecurrente y está definida de la siguiente manera:

def crear_mensajes(grafo_final, grapho, datos=[], lista_mensajes=[], lista_arcos=[], combinaciones_arcos=[])

...

return lista_arcos, lista_mensajes

Los parámetros de entrada son:

- grafo_final. Es la lista de implementaciones de servicio elegida para la composición de la aplicación actual.
- grapho. Es la selección de servicios pedidos por cliente, formando un grafo según sus requerimientos.
- datos. Es una variable interna auxiliar que ayuda a conocer información del servicio del que proviene el mensaje. Es para su uso en la recursividad de la función.
- lista_mensajes. Es la lista final de objetos de la clase Mensajes, es decir, son los mensajes existentes entre las implementaciones finales elegidas.. También es un parámetro de salida.
- lista_arcos. Es una lista de tuplas que almacena parejas de servicios. Estas parejas dan la información de los orígenes y destinos de los mensajes creados. Es para uso interno y fuera de la función no tiene sentido. También es un parámetro de salida usado las veces que las función es recursiva.
- combinaciones_arcos. Inicialmente está vacía, pero se trata de una lista que almacena todas las combinaciones posibles del grafo solicitado por el cliente. Su funcionalidad es tener en tuplas las posibles opciones de mensajes de red para no crear dos iguales del mismo enlace entre dos servicios.

Los parámetros de salida son *lista_arcos* y *lista_mensajes*. Ambos son devueltos en cada devolución interna de la función, pero finalmente, cuando todos los mensajes están creados, *lista_arcos* es irrelevante y no se utiliza.

5.4.3.7 planificacion_red

La función *planificacion_red* tiene como misión calcular los ciclos elementales que tarda cada uno de lo mensajes en transmitirse y calcula el tiempo de respuesta de la aplicación teniendo en cuenta también los ciclos elementales que tardan las tareas en ejecutarse. Además, siguiendo el paradigma FTT, la transmisión de un mensaje o la ejecución de una tarea no empieza hasta el ciclo elemental siguiente al que haya finalizado el proceso anterior. Viene definida por:

def planificacion_red(caso, lista_mensajes, grafo_final, lista_caminos)

...

return R_App, True/False

Los parámetros de entrada son:

- **caso**. Este parámetro identifica si la planificación se debe hacer para RTA o para RUB, ejecutando su parte de código correspondiente. Es decir, dependiendo del test de planificabilidad que se esté ejecutando en ese momento, esta función llamará a la función *paralelos_mensajes_RTA* o *paralelos_mensajes_RUB*.
- **lista_mensajes**. Es la lista de los mensajes de red con los que se comunican las implementaciones de servicios que forman la aplicación.
- **grafo_final**. Es la lista de implementaciones de servicio que forman la aplicación solicitada por el cliente.
- **lista_caminos**. Es la lista de caminos posibles dentro del grafo solicitado por el cliente, y su cometido es no dejar de explorar ningún posible camino al planificar la red.

Y los parámetros de salida son:

- **R_App**. Es el tiempo de respuesta de la aplicación. En este tiempo se tiene en cuenta que un proceso termina a mitad de un ciclo elemental, el mensaje generado después no empieza hasta el siguiente ciclo elemental. Este tiempo, junto con las implementaciones de servicio que componen la aplicación, es el objetivo final.
- **True/False**. Es una variable booleana que es *True* en caso de que la planificación de la red sea correcta y *False* en caso contrario.

5.4.3.8 paralelos_mensajes_RTA

La función *paralelos_mensajes* calcula el tiempo de respuesta de la aplicación a partir de las implementaciones de servicio que la componen y los mensajes con los que se comunican dichas implementaciones. Para ello, se apoya en una serie de bucles que analizan el grafo solicitado por el cliente y va calculando el tiempo de respuesta para cada uno de los caminos paralelos del grafo. Finalmente se queda con el mayor. Su definición es:

def paralelos_mensajes_RTA(lista_mensajes, grafo_final, lista_caminos)
...
return max(tiempos_paralelo_mensaje)

Los parámetros de entrada son:

- **lista_mensajes**. Es lista de los mensajes con los que se comunican las implementaciones elegidas para componer la aplicación.
- **grafo_final**. Las implementaciones de servicio elegidas para componer la aplicación.
- **lista_caminos**. El conjunto de caminos posibles para recorrer el grafo solicitado por el cliente.

El parámetro de salida es el tiempo de respuesta de la aplicación.

5.4.3.9 paralelos_mensajes_RUB

La función *paralelos_mensajes_RUB* hace exactamente lo mismo que *paralelos_mensajes_RTA*, pero usando los tiempos RUB almacenados en las implementaciones. Su finalidad es simplemente separar las funciones que se encargan de la planificabilidad RTA de las funciones que se encargan de la planificabilidad RUB. Viene definida de la siguiente forma:

def paralelos_mensajes_RUB(lista_mensajes, grafo_final, lista_caminos)

...

return max(tiempos_paralelo_mensaje)

5.4.3.10 poda_por_solicitud

La función *poda_por_solicitud* tiene como objetivo eliminar del grafo de servicios aquellos que no sean solicitados por el cliente, con el fin de tener un árbol de exploración reducido para el caso particular de cada aplicación y tener una eficiencia mayor en cuanto al tiempo. La función queda definida de la siguiente forma:

def poda_por_solicitud(grafo_ofrecido_ordenado, grafo_solicitado)

...

return grafo_ofrecido_ordenado_final, servicios_pedidos

Donde los parámetros de entrada son el grafo de servicios y grafo solicitado por el cliente, y los parámetros de salida son el nuevo grafo sin los servicios no útiles y una lista con los servicios finales requeridos por el cliente.

5.4.3.11 filtro_por_rub

La función *filtro_por_rub* hace, como su nombre indica, un filtrado previo mediante el test de planificabilidad RUB. Para ello, planifica el sistema con cada una de las implementaciones del grafo ofrecido y comprueba, de manera individual, si el sistema es planificable. En caso de no ser planificable, elimina la implementación correspondiente ya que si no es planificable de manera individual, no lo es formando parte de una combinación. La función queda definida así:

def filtro_por_rub(grafo_ofrecido_ordenado, tareas_por_nodo)

...

return grafo_ofrecido_ordenado

Los parámetros de entrada son grafo_ofrecido_ordenado, que contiene todas las implementaciones de servicio existentes; y tareas_por_nodo, que contiene las tareas existentes en cada nodo para poder hacer la planificación RUB. El único parámetro de salida es el grafo con las implementaciones que han pasado el filtro.

Cabe destacar que esta función solo se ejecuta para la prueba número 3.

5.4.4. Funciones relativas a la obtención de resultados

5.4.4.1 ajustar_tiempos_EC

La función *ajustar_tiempos_EC* se ocupa de hacer una división que adapta el tiempo de respuesta de las tareas al formato en ciclos elementales. Para ellos, analiza cada tarea e implementación de servicio y divide su tiempo de respuesta entre el valor del ciclo elemental. Por tanto, su único parámetro de entrada y salida es el diccionario que contiene todas las tareas previas e implementaciones de servicio:

def ajustar_tiempos_EC(tareas_en_nodos)

...

return tareas_en_nodos

5.4.4.2 ajustar_tiempos_RUB

La función *ajustar_tiempos_RUB* hace lo mismo que *ajustar_tiempos_EC*, pero aplicado a los tiempos RUB de las implementaciones. Como en el caso de *paralelos_mensajes_RUB*, su finalidad no es otra que separar las funciones que se emplean para cada uno de los tests de planificabilidad RTA y RUB.

def ajustar_tiempos_RUB(tareas_en_nodos)

...

return tareas_en_nodos

5.4.4.3 informe

La función *informe* genera una documentación para ser vista por pantalla en tiempo de ejecución. Se trata de un resumen de los resultados obtenidos.

def informe(tareas_por_nodos, grafo_ofrecido_ordenado_final)

...

5.4.4.4 resultados

La función *resultados* es la encargada de generar los documentos necesarios sobre los resultados obtenidos. Genera el tiempo de ejecución del programa, y crea los ficheros con los datos de tiempo de ejecución, tiempos en ciclos elementales de la aplicación, número de combinaciones posibles para explorar, así como la combinación de implementaciones elegidas para la aplicación dada.

def resultados(tiempo_respuesta, camino_final)

...

5.5. Conclusiones

Las decisiones de diseño tomadas en cada una de las funciones ha sido la que en cada momento se ha considerado la más adecuada. Sin embargo, puesto que se trata de un diseño recursivo, las decisiones tomadas en segundo término han afectado de manera significativa a las decisiones de diseño tomadas inicialmente, modificando funciones ya escritas y provocando nuevas complicaciones.

De hecho, después de tener gran parte del código, éste ha tenido que ser modificado para adaptarse a su disposición final. Esto se debe a que se ha ido programando por partes. Se programó, en primer lugar, la búsqueda exhaustiva de las combinaciones de implementaciones según si cumple el tiempo de respuesta permitido a partir de su tiempo de ejecución en el peor caso; en segundo lugar, la comprobación de la utilización de los nodos de la red para cada combinación; y por último, la comprobación de la planificación para cada combinación válida. Después de desarrollar todo esto, se reescribió para que compruebe cada combinación a medida que se encuentra, es decir, el algoritmo localiza una combinación, comprueba la utilización de los nodos para dicha combinación, y confirma que es planificable. En caso de que alguna combinación no cumpla el tiempo permitido, sature algún nodo, o no sea planificable, es descartada directamente.

Si bien antes de la modificación el algoritmo era más rápido para casos de árboles de exploración pequeños, tras ésta, es más rápido para situaciones más complicadas.

Por tanto, esta modificación supuso un ahorro en lineas de código y además, se consiguió un algoritmo más eficiente.

En resumen, este capítulo debería servir de manual para que un usuario con conocimientos de *Python* pueda ser capaz de abordar el programa desarrollado para su interpretación, uso, modificación y mejora.

Capítulo 6

Evaluación y resultados

En este capítulo se muestran los resultados obtenidos al evaluar los algoritmos programados. Se realizarán una serie de pruebas, con distintas condiciones iniciales, a partir de las cuales se van obtener las características de funcionamiento de cada algoritmo. Finalmente, se hará una comparativa para asegurar el correcto funcionamiento y observar las diferencias entre las prestaciones para cada caso.

6.1. Introducción

En esta parte del documento se recoge la información obtenida al realizar una serie de pruebas sobre los dos algoritmos implementados en el proyecto. Los resultados obtenidos son, entre otros, tiempos de respuesta o falsos negativos tras una planificación extremo a extremo.

6.1.1. Definición de las pruebas

Para obtener resultados que ayuden a entender el funcionamiento de los algoritmos, así como para conocer su punto óptimo de trabajo, se han desarrollado tres pruebas diferentes que se han ejecutado para diferentes condiciones definidas previamente. Estas condiciones iniciales que afectan a los resultados son el grafo seleccionado por el cliente, la carga previa de cada nodo, y el número de implementaciones existentes para cada servicio. De esta última condición se desprende que uno de los resultados a analizar es el número de combinaciones a explorar por los algoritmos, que viene dado por la siguiente fórmula:

$$num_{comb} = \prod I_i \tag{6.1}$$

donde num_{comb} es el número de combinaciones posibles, e I_i es el número de implementaciones efectivas del servicio i. Se deduce a partir de la ecuación 6.1 que el número de combinaciones crece acorde al número de implementaciones de servicio. También cabe destacar que esta variable es útil en cuanto al

coste computacional que pueden conseguir los algoritmos, ya que, el algoritmo heurístico explora un menor número de combinaciones.

La ***prueba 1*** que se ha aplicado al programa es ejecutar en modo exhaustivo y en modo heurístico, a partir de las condiciones iniciales mencionadas anteriormente:

- **Algoritmo utilizado**. Puede ser exhaustivo o heurístico.
- **Grafo solicitado**. Es la combinación de servicios seleccionados por el cliente en un determinado orden.
- **Carga previa de los nodos**. Formada por las tareas previas, la utilización previa influye en la búsqueda de implementaciones para una aplicación dada.
- **Número de implementaciones por servicio**. Este número influye en el número total de exploraciones.

Los principales resultados que se persiguen con la ejecución de esta prueba son:

- Consecución del tiempo que tarda cada algoritmo en encontrar una combinación óptima o subóptima. De esta forma, se puede hacer una comparativa de qué algoritmo es mejor dada una determinada situación con unas determinadas condiciones.
- Obtención de conclusiones de cómo afecta la carga previa del nodo a la búsqueda de la combinación óptima/subóptima.
- Obtención de conclusiones a partir de los porcentajes de error obtenidos por utilizar un algoritmo heurístico.

La ***prueba 2*** consiste en utilizar la cota superior al tiempo de respuesta que proporciona el test de planificabilidad RUB, como sustituto del tiempo de respuesta dado por el test RTA. La prueba consta de tres partes:

- Primero se tienen en cuenta los **falsos negativos** generados por el uso de ambos de tests de planificabilidad, es decir, los casos en los que los tests utilizados afirman que no existe solución cuando en realidad si que la hay. De esta parte de pueden extraer conclusiones de en qué condiciones en más fiable el uso de test de planificabilidad RUB.
- La segunda parte consiste en utilizar la cota superior al tiempo de respuesta, calculada mediante el **test RUB, como tiempo de respuesta** propiamente dicho. Así, se puede comprobar hasta que punto el test de planificabilidad RUB realmente mejora las prestaciones de su equivalente exhaustivo. Por tanto, esta prueba tiene como objeto determinar cómo el uso de RUB para el cálculo del tiempo de respuesta afecta no sólo a un nodo concreto, sino al tiempo de respuesta extremo a extremo estimado de la aplicación.

- Finalmente, la prueba hace un **comparativa de los tiempos** empleados en conseguir los resultados. El tiempo de ejecución del programa no es el mismo al ejecutar el test RTA o el test RUB. De esta forma, se puede obtener conclusiones del tiempo que se tarda en conseguir los resultados.

La ***prueba 3*** se basa en un **control de admisión** regulado por el test de planificabilidad RUB. Esta prueba comprobará primero si las tareas son planificables individualmente en cada nodo físico a partir del tiempo dado por el test RUB y, en caso negativo, descartarlas del conjunto de implementaciones de servicio disponibles para la composición de la aplicación, reduciendo así el número de combinaciones finales y el coste computacional del sistema final.

Las conclusiones a obtener de esta prueba son ver si esta reducción del tiempo y coste computacional afecta de manera directa en las prestaciones del sistema.

6.2. Resultados obtenidos

En este apartado se exponen los resultados obtenidos para cada una de las pruebas realizadas.

6.2.1. Prueba 1

La prueba 1 se realiza sobre tres tipos de aplicaciones diferentes, dos simples (una con tres servicios y otra con cinco servicios) y una compleja. El plazo de la aplicación para todos los casos es de 1500ms. La red está compuesta por 4 nodos físicos, cada uno de ellos con 4 tareas previas alojadas en cada uno de ellos, provocando una carga previa de 0 %, 30 % o 55 %. Este porcentaje de utilización influye, como era de esperar, en los resultados obtenidos. Estos resultados son en media, conseguidos tras 10 ejecuciones para cada caso.

6.2.1.1. Aplicación sencilla. Tres servicios en serie

Se selecciona una aplicación simple, con servicios A, B y C colocados en serie, como se puede ver en la figura 6.1.

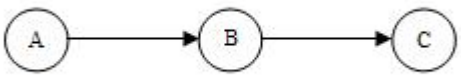

Figura 6.1: Aplicación sencilla con tres servicios en serie

- Teniendo en cuenta que existan **4 implementaciones para cada servicio**, con una **utilización previa de 0 %** (ver tabla 6.1), obtenemos estos resultados:

- Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_1, B_2, C_0], con un tiempo final de 6 EC's.
- Tras la **búsqueda heurística**, la solución encontrada es [A_1, B_0, C_2], con un tiempo final de 6 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	64	1.457	0,041	0	6	-
Heurístico	8	0,261	0,020	0	6	0

Tabla 6.1: Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 4 implementaciones por servicio y carga previa 0 %

- Teniendo en cuenta que existan **4 implementaciones para cada servicio**, con una **utilización previa de 30 %** (ver tabla 6.2), obtenemos estos resultados:
  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_1, B_1, C_0], con un tiempo final de 10 EC's.
  - Tras la **búsqueda heurística**, la solución encontrada es [A_1, B_1, C_0], con un tiempo final de 10 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	64	0,723	0,060	30	10	-
Heurístico	8	0,235	0,034	30	10	0

Tabla 6.2: Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 4 implementaciones por servicio y carga previa 30 %

- Teniendo en cuenta que existan **4 implementaciones para cada servicio**, con una **utilización previa de 55 %** (ver tabla 6.3), obtenemos estos resultados:
  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_1, B_0, C_0], con un tiempo final de 10 EC's.
  - Tras la **búsqueda heurística**, la solución encontrada es [A_1, B_0, C_0], con un tiempo final de 10 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	64	2.357,2	0,585	55	10	-
Heurístico	8	0,831	0,053	55	10	0

Tabla 6.3: Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 4 implementaciones por servicio y carga previa 55 %

- Teniendo en cuenta que existan **6 implementaciones para cada servicio**, con una **utilización previa de 0 %** (ver tabla 6.4), obtenemos estos resultados:

  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_2, B_2, C_2], con un tiempo final de 6 EC's.
  - Tras la **búsqueda heurística**, la solución encontrada es [A_2, B_2, C_2], con un tiempo final de 6 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	216	5.153,2	0,735	0	6	-
Heurístico	8	0,226	0,019	0	6	0

Tabla 6.4: Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 6 implementaciones por servicio y carga previa 0 %

- Teniendo en cuenta que existan **6 implementaciones para cada servicio**, con una **utilización previa de 30 %** (ver tabla 6.5), obtenemos estos resultados:

  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_2, B_1, C_0], con un tiempo final de 9 EC's.
  - Tras la **búsqueda heurística**, la solución encontrada es [A_2, B_0, C_2], con un tiempo final de 10 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	216	18.621,5	2,792	30	9	-
Heurístico	8	0,634	0,025	30	10	11,1

Tabla 6.5: Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 6 implementaciones por servicio y carga previa 30 %

- Teniendo en cuenta que existan **6 implementaciones para cada servicio**, con una **utilización previa de 55 %** (ver tabla 6.6), obtenemos estos resultados:

  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, no consiguiendo ninguna implementación [Ø] para dicha solución. Este problema viene dado porque en algunos casos las tareas previas con estas características no son planificables y porque en otros casos no es planificable el sistema con una posible solución dada.
  - Tras la **búsqueda heurística**, no se encuentra solución, [Ø]. En este caso, todas las pruebas son erróneas porque no existe la planificación a causa de las tareas previas del sistema.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	216	4,665	0,894	55	-	-
Heurístico	8	0,486	0,063	55	-	-

Tabla 6.6: Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 6 implementaciones por servicio y carga previa 55 %

- Teniendo en cuenta que existan **8 implementaciones para cada servicio**, con una **utilización previa de 0 %** (ver tabla 6.7), obtenemos estos resultados:

  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_0, B_4, C_5], con un tiempo final de 6 EC's.
  - Tras la **búsqueda heurística**, la solución encontrada es [A_0, B_4, C_5], con un tiempo final de 6 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	512	10.983,3	1,178	0	6	-
Heurístico	8	0,221	0,014	0	6	0

Tabla 6.7: Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 8 implementaciones por servicio y carga previa 0 %

- Teniendo en cuenta que existan **8 implementaciones para cada servicio**, con una **utilización previa de 30 %** (ver tabla 6.8), obtenemos estos resultados:

  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_0, B_4, C_5], con un tiempo final de 11 EC's. En algunos casos las tareas previas no son planificables; a pesar de ello, se encuentra la solución óptima.
  - Tras la **búsqueda heurística**, la solución encontrada es [A_0, B_4, C_5], con un tiempo final de 11 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	512	4.803,5	0,935	30	11	-
Heurístico	8	0,620	0,012	30	11	0

Tabla 6.8: Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 8 implementaciones por servicio y carga previa 30 %

- Teniendo en cuenta que existan **8 implementaciones para cada servicio**, con una **utilización previa de 55 %** (ver tabla 6.9), obtenemos estos resultados:

- Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_0, B_4, C_3], con un tiempo final de 13 EC's. En el caso de las últimas exploraciones del árbol, las combinaciones seleccionadas no son posibles por utilización, es decir, se supera la utilización máxima permitida para un nodo.
- Tras la **búsqueda heurística**, la solución encontrada es [A_0, B_4, C_3], con un tiempo final de 13 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	512	3.355,9	1,064	55	13	-
Heurístico	8	0,632	0,030	55	13	0

Tabla 6.9: Resultados relativos a la prueba 1. Para una aplicación con 3 servicios en serie, 8 implementaciones por servicio y carga previa 55 %

6.2.1.2. Aplicación sencilla. Cinco servicios en serie

Se selecciona una aplicación simple, con servicios A, B1, C, B2 y D colocados en serie, como se ve en al figura 6.2.

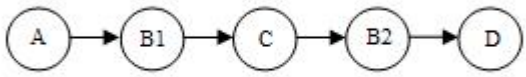

Figura 6.2: Aplicación sencilla con cinco servicios en serie

- Teniendo en cuenta que existan **4 implementaciones para cada servicio**, con una **utilización previa de 0 %** (ver tabla 6.10), obtenemos estos resultados:
  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_1, B_0, C_2, B_3, D_3], con un tiempo final de 16 EC's.
  - Tras la **búsqueda heurística**, la solución encontrada es [A_3, B_1, C_2, B_0, D_3], con un tiempo final de 16 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	384	9.785,8	0,562	0	16	-
Heurístico	8	0,365	0,023	0	16	0

Tabla 6.10: Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 4 implementaciones por servicio y carga previa 0 %

- Teniendo en cuenta que existan **4 implementaciones para cada servicio**, con una **utilización previa de 30 %** (ver tabla 6.11), obtenemos estos resultados:

  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, no consiguiendo ninguna implementación [Ø] para dicha solución.
  - Tras la **búsqueda heurística**, no se encuentra solución, [Ø]. En este caso, todas las pruebas son erróneas porque no existe la planificación a causa de las tareas previas del sistema.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	384	2,810	0,589	30	-	-
Heurístico	8	0,118	0,015	30	-	-

Tabla 6.11: Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 4 implementaciones por servicio y carga previa 30 %

- Teniendo en cuenta que existan **4 implementaciones para cada servicio**, con una **utilización previa de 55 %** (ver tabla 6.12), obtenemos estos resultados:

  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, no consiguiendo ninguna implementación [Ø] para dicha solución.
  - Tras la **búsqueda heurística**, no se encuentra solución, [Ø]. En este caso, todas las pruebas son erróneas porque no existe la planificación a causa de las tareas previas del sistema.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	384	4,562	0,705	55	-	-
Heurístico	8	0,185	0,015	55	-	-

Tabla 6.12: Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 4 implementaciones por servicio y carga previa 55 %

- Teniendo en cuenta que existan **6 implementaciones para cada servicio**, con una **utilización previa de 0 %** (ver tabla 6.13), obtenemos estos resultados:

  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_2, B_2, C_2, B_0, D_4], con un tiempo final de 16 EC's.
  - Tras la **búsqueda heurística**, la solución encontrada es [A_2, B_2, C_3, B_0, D_0], con un tiempo final de 17 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	3240	88.594,3	3,943	0	16	-
Heurístico	8	0,356	0,015	0	17	6,3

Tabla 6.13: Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 6 implementaciones por servicio y carga previa 0 %

- Teniendo en cuenta que existan **6 implementaciones para cada servicio**, con una **utilización previa de 30 %** (ver tabla 6.14), obtenemos estos resultados:
  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_0, B_0, C_1, B_1, D_0], con un tiempo final de 20 EC's.
  - Tras la **búsqueda heurística**, la solución encontrada es [A_2, B_2, C_2, B_0, D_0], con un tiempo final de 23 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	3240	252.829,2	16,324	30	20	-
Heurístico	8	0,726	0,027	30	23	15

Tabla 6.14: Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 6 implementaciones por servicio y carga previa 30 %

- Teniendo en cuenta que existan **6 implementaciones para cada servicio**, con una **utilización previa de 55 %** (ver tabla 6.15), obtenemos estos resultados:
  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, no consiguiendo ninguna implementación [Ø] para dicha solución. Este problema viene dado en los primeros casos por las tareas previas, que con estas características no son planificables; y en los casos finales porque el nodo tiene una carga mayor a la unidad.
  - Tras la **búsqueda heurística**, no se encuentra solución, [Ø]. En este caso, todas las pruebas son erróneas porque no existe la planificación a causa de las tareas previas del sistema.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	3240	9,172	2,080	55	-	-
Heurístico	8	0,568	0,033	55	-	-

Tabla 6.15: Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 6 implementaciones por servicio y carga previa 55 %

- Teniendo en cuenta que existan **8 implementaciones para cada servicio**, con una **utilización previa de 0 %** (ver tabla 6.16), obtenemos estos resultados:

- Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_0, B_4, C_5, B_5, D_7], con un tiempo final de 16 EC's. Las últimas combinaciones provocan una utilización del nodo superior a la unidad, por tanto son casos no evaluables.
- Tras la **búsqueda heurística**, la solución encontrada es [A_0, B_4, C_5, B_5, D_7], con un tiempo final de 16 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	14336	358.852,6	32,790	0	16	-
Heurístico	8	0,338	0,026	0	16	0

Tabla 6.16: Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 8 implementaciones por servicio y carga previa 0 %

- Teniendo en cuenta que existan **8 implementaciones para cada servicio**, con una **utilización previa de 30 %** (ver tabla 6.17), obtenemos estos resultados:
  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_0, B_4, C_5, B_5, D_0], con un tiempo final de 25 EC's. En algunos casos las tareas previas no son planificables; a pesar de ello, se encuentra la solución óptima.
  - Tras la **búsqueda heurística**, la solución encontrada es [A_0, B_4, C_5, B_5, D_7], con un tiempo final de 26 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	14336	110.305	7,500	30	25	-
Heurístico	8	0,688	0,040	30	26	4

Tabla 6.17: Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 8 implementaciones por servicio y carga previa 30 %

- Teniendo en cuenta que existan **8 implementaciones para cada servicio**, con una **utilización previa de 55 %** (ver tabla 6.18), obtenemos estos resultados:
  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_0, B_4, C_3, B_5, D_7], con un tiempo final de 31 EC's. Las últimas combinaciones provocan una utilización del nodo superior a la unidad, por tanto son casos no evaluables.
  - Tras la **búsqueda heurística**, la solución encontrada es [A_0, B_4, C_3, B_5, D_7], con un tiempo final de 31 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	14336	18.354,7	1,567	55	31	-
Heurístico	8	0,763	0,034	55	31	0

Tabla 6.18: Resultados relativos a la prueba 1. Para una aplicación con 5 servicios en serie, 8 implementaciones por servicio y carga previa 55 %

6.2.1.3. Aplicación compleja. Cinco servicios con paralelos

Se selecciona una aplicación compleja, con servicios A, B, C y D colocados como se muestra en la figura 6.3.

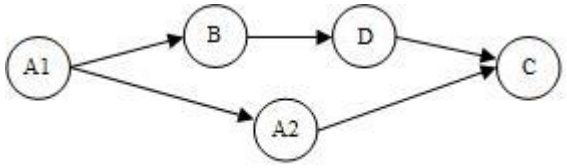

Figura 6.3: Aplicación compleja de cinco servicios

- Teniendo en cuenta que existan **4 implementaciones para cada servicio**, con una **utilización previa de 0 %** (ver tabla 6.19), obtenemos estos resultados:
  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_3, B_1, D_2, A_0, C_2], con un tiempo final de 12 EC's.
  - Tras la **búsqueda heurística**, la solución encontrada es [A_1, B_0, D_3, A_3, C_2], con un tiempo final de 13 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	384	12.233,8	1,195	0	12	-
Heurístico	8	0,314	0,039	0	13	8,3

Tabla 6.19: Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 4 implementaciones por servicio y carga previa 0 %

- Teniendo en cuenta que existan **4 implementaciones para cada servicio**, con una **utilización previa de 30 %** (ver tabla 6.20), obtenemos estos resultados:
  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_1, B_1, D_0, A_3, C_0], con un tiempo final de 19 EC's.

- Tras la **búsqueda heurística**, la solución encontrada es [A_1, B_1, D_2, A_3, C_0], con un tiempo final de 20 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	384	4.791,4	1,243	30	19	-
Heurístico	8	0,479	0,025	30	20	5,3

Tabla 6.20: Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 4 implementaciones por servicio y carga previa 30 %

- Teniendo en cuenta que existan **4 implementaciones para cada servicio**, con una **utilización previa de 55 %** (ver tabla 6.21), obtenemos estos resultados:

  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, no consiguiendo implementaciones, [Ø]. En este caso, todas las pruebas son erróneas porque no existe la planificación a causa de las tareas previas del sistema.
  - Tras la **búsqueda heurística**, no se encuentra solución, [Ø].

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	384	6,240	0,981	55	-	-
Heurístico	8	0,281	0,023	55	-	-

Tabla 6.21: Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 4 implementaciones por servicio y carga previa 55 %

- Teniendo en cuenta que existan **6 implementaciones para cada servicio**, con una **utilización previa de 0 %** (ver tabla 6.22), obtenemos estos resultados:

  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_2, B_2, D_0, A_0, C_1], con un tiempo final de 12 EC's.
  - Tras la **búsqueda heurística**, la solución encontrada es [A_2, B_2, D_0, A_5, C_2], con un tiempo final de 14 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	3240	94.745	9,995	0	12	-
Heurístico	8	0,286	0,020	0	14	16,7

Tabla 6.22: Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 6 implementaciones por servicio y carga previa 0 %

- Teniendo en cuenta que existan **6 implementaciones para cada servicio**, con una **utilización previa de 30 %** (ver tabla 6.23), obtenemos estos resultados:

  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_2, B_0, D_0, A_0, C_1], con un tiempo final de 15 EC's.
  - Tras la **búsqueda heurística**, la solución encontrada es [A_2, B_0, D_0, A_5, C_2], con un tiempo final de 20 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	3240	275.183,4	24,324	30	15	-
Heurístico	8	0.776	0,038	30	20	33,4

Tabla 6.23: Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 6 implementaciones por servicio y carga previa 30 %

- Teniendo en cuenta que existan **6 implementaciones para cada servicio**, con una **utilización previa de 55 %** (ver tabla 6.24), obtenemos estos resultados:

  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, no consiguiendo ninguna implementación [Ø] para dicha solución.
  - Tras la **búsqueda heurística**, no se encuentra solución, [Ø].

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	3240	9,799	2,377	55	-	-
Heurístico	8	0,591	0,034	55	-	-

Tabla 6.24: Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 6 implementaciones por servicio y carga previa 55 %

- Teniendo en cuenta que existan **8 implementaciones para cada servicio**, con una **utilización previa de 0 %** (ver tabla 6.25), obtenemos estos resultados:

  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_5, B_4, D_7, A_1, C_5], con un tiempo final de 12 EC's. Las últimas combinaciones provocan una utilización del nodo superior a la unidad, por tanto son casos no evaluables.
  - Tras la **búsqueda heurística**, la solución encontrada es [A_0, B_4, D_7, A_5, C_5], con un tiempo final de 13 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	14336	436.536,8	40,272	0	12	
Heurístico	8	0,310	0,031	0	13	8,3

Tabla 6.25: Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 8 implementaciones por servicio y carga previa 0 %

- Teniendo en cuenta que existan **8 implementaciones para cada servicio**, con una **utilización previa de 30 %** (ver tabla 6.26), obtenemos estos resultados:

  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_0, B_4, D_7, A_5, C_5], con un tiempo final de 23 EC's. En algunos casos las tareas previas no son planificables; a pesar de ello, se encuentra la solución óptima.
  - Tras la **búsqueda heurística**, la solución encontrada es [A_0, B_4, D_7, A_5, C_5], con un tiempo final de 23 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	14336	130.387,2	7,893	30	23	-
Heurístico	8	0.755	0,029	30	23	0

Tabla 6.26: Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 8 implementaciones por servicio y carga previa 30 %

- Teniendo en cuenta que existan **8 implementaciones para cada servicio**, con una **utilización previa de 55 %** (ver tabla 6.27), obtenemos estos resultados:

  - Primero se hace la **búsqueda exhaustiva** para encontrar la solución óptima, consiguiendo las implementaciones [A_0, B_4, D_7, A_5, C_3], con un tiempo final de 28 EC's. Alguna combinaciones no son posibles porque la planificación no es posible a causa de las tareas previas. Las últimas combinaciones provocan una utilización del nodo superior a la unidad, y por lo tanto, son casos no evaluables.
  - Tras la **búsqueda heurística**, la solución encontrada es [A_0, B_4, D_7, A_5, C_3], con un tiempo final de 28 EC's.

	Num_{comb}	Tiempo (ms)	Desviación σ	Carga del nodo (%)	EC's	Error (%)
Exhaustivo	14336	39.549,2	5,247	55	28	-
Heurístico	8	0,780	0,038	55	28	0

Tabla 6.27: Resultados relativos a la prueba 1. Para una aplicación compleja de 5 servicios, 8 implementaciones por servicio y carga previa 55 %

6.2.1.4. Conclusiones

De las tablas anteriores se desprende que a medida que aumenta el número de implementaciones por cada servicio, el tiempo que se tarda en encontrar una solución optima usando el método exhaustivo aumenta, es decir, el tiempo que se tarda en explorar todo el árbol es mayor conforme mayor sea dicho árbol. Sin embargo, con el algoritmo heurístico, el tiempo está por debajo de 1ms.

Este tiempo conseguido a través del algoritmo exhaustivo también va en aumento a medida que la carga previa de los nodos aumenta también. Por tanto, para carga de nodo medias los tiempos son considerablemente mayores y en el caso de cargas de nodo altas el tiempo asciende a minutos, algo bastante perjudicial para la efectividad del sistema completo. En este último caso, algunas combinaciones provocan una utilización del nodo superior a la unidad, descartándose y ahorrando tiempo, por lo que se encuentra una solución óptima en un tiempo menor al esperado.

Al margen del tiempo que se tarda en encontrar una solución óptima, el tiempo de respuesta en ciclos elementales también aumenta a medida que existen mayor número de implementaciones disponibles por cada servicio.

En cualquier caso, el algoritmo heurístico siempre consigue tiempos menores a un milisegundo. Además, los tiempos de respuesta en ciclos elementales de las combinaciones subóptimas son, en la mayoría de los casos iguales o un poco superiores (del orden de 2EC's), sin variar en gran medida y en la mayoría de los casos la combinación seleccionada como subóptima.

En resumen, podemos concluir que si trabajamos con el algoritmo exhaustivo, los tiempos para encontrar la solución adecuada no serán muy grandes para pocas implementaciones por cada servicio, o en su defecto, para cargas previas medias-altas de los nodos de la red.

Por otro lado, se puede afirmar que con el uso de algoritmo heurístico encontramos una notable mejora en el tiempo de la consecución de la solución subóptima, bastante semejante en cuanto a las implementaciones seleccionadas y los tiempos de respuesta conseguidos con respecto a su homólogo exhaustivo. Además, el coste computacional es, en todos los casos, inferior que en los casos exhaustivos y dentro de un rango conocido, lo que proporciona mayor estabilidad al sistema.

En cuanto al parámetro del número de combinaciones existentes, sirve para destacar cómo afecta la poda heurística al coste computacional, y cómo éste está relacionado intrínsecamente con la variedad de caminos a explorar en un árbol.

6.2.2. Prueba 2

La *prueba* 2, como se explicó en la sección anterior, consta de tres partes, y se basa en el uso de test de planificabilidad RUB para analizar el sistema. Para llevar a cabo la prueba, se calculan los falsos negativos del test de planificabilidad RUB, es decir, se analiza las veces que dicha planificación ha declarado que no puede planificarse el sistema con las implementaciones seleccionadas y luego realmente éste sí que era planificable. Después, a raíz de los resultados obtenidos, se analiza si se puede utilizar el tiempo dado por

el test de planificabilidad RUB como sustituto del tiempo de respuesta dado por el test de planificabilidad RTA para una aplicación dada, consiguiendo unas prestaciones semejantes, y no introduciendo mucho error. Finalmente, se realiza un análisis de los tiempos que tarda en ejecutarse el programa para encontrar los resultados, es decir, se comparan y estudian los tiempos de ejecución del programa desarrollado.

De nuevo, los resultados obtenidos se obtienen de realizar la media tras 10 iteraciones.

6.2.2.1. Aplicación sencilla. Tres servicios en serie

La primera ejecución se realiza sobre una aplicación sencilla, compuesta por tres servicios en serie, "A", "B", y "C". El plazo de la aplicación fijado es de 5000ms. Hay que tener en cuenta que el número de combinaciones existentes para cada caso varía dependiendo del número de implementaciones por servicio y del número de servicios usados en la aplicación; para este caso, en el que los resultados se han obtenido a partir de 4, 6 y 8 implementaciones, el número de combinaciones es de 64, 216 y 512, respectivamente.

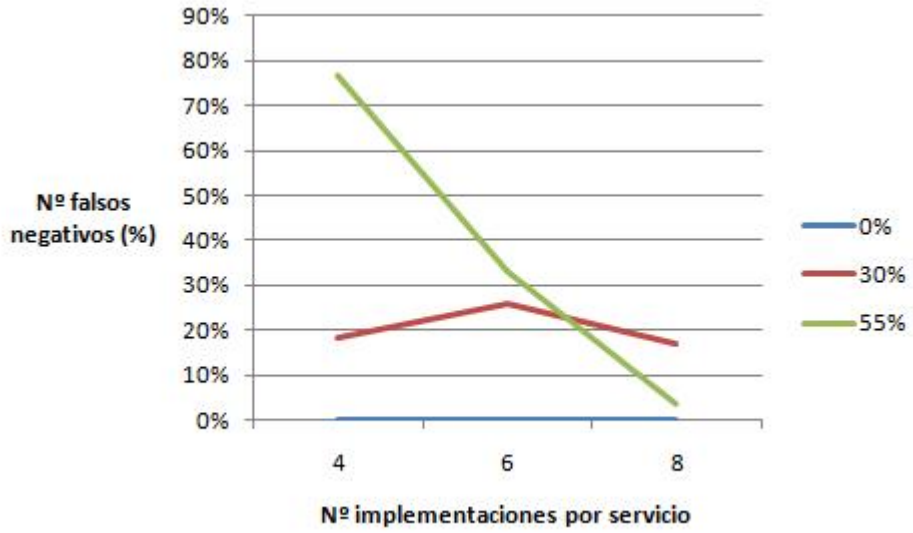

Figura 6.4: Falsos negativos relativos a una aplicación sencilla de tres servicios

Según se observa en la gráfica 6.4, la carga del nodo influye de manera directa en la cantidad de falsos negativos generados, siendo su número creciente al aumentar el porcentaje de utilización de los nodos. Sin embargo, como se observa en el caso de utilización previa de los nodos al 55 %, existe una excepción, y ésta se da aumentar el número de implementaciones por servicio. Al tratar con utilizaciones altas, es más fácil que el sistema no sea planificable por superar la unidad. Esto hace que los casos de falsos negativos disminuyan; no porque su test de planificabilidad lo detecte, sino porque existen, en realidad, menos casos a analizar. En este ejemplo, el caso más afectado por este hecho es al ejecutar con 8 implementaciones

por servicio.

Además, cuando se produce este fenómeno, el test de planificabilidad RUB no encuentra combinaciones óptimas para la aplicación.

		4 Imp/Servicio (EC's)	6 Imp/Servicio (EC's)	8 Imp/Servicio (EC's)
Nodos al 0 %	RTA	6	6	6
	RUB	6	6	6
	Error (%)	0	0	0
Nodos al 30 %	RTA	10 (9,6)	10 (9,5)	10 (9,8)
	RUB	11 (10,1)	10 (9,7)	11 (10,4)
	Error (%)	10	0	10
Nodos al 55 %	RTA	12 (11,5)	12 (11,3)	12 (11,1)
	RUB	0	27 (26,5)	25
	Error (%)	-	125	108,33

Tabla 6.28: Datos relativos a los tiempos de respuesta y error introducido de una aplicación sencilla con tres servicios en serie

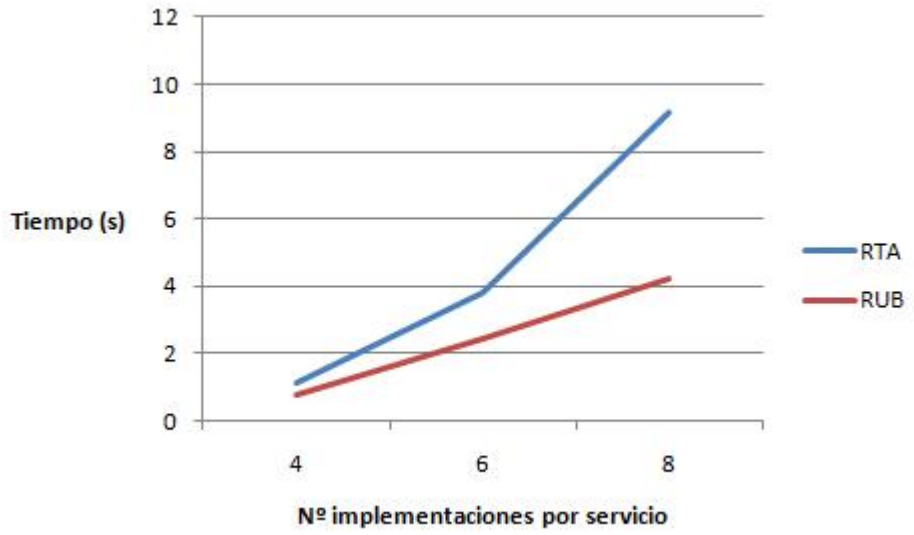

Figura 6.5: Tiempo de ejecución. App. sencilla de tres servicios y utilización previa al 0 %

En cuanto al uso del tiempo obtenido por el test de planificabilidad RUB como tiempo de respuesta de la aplicación, se desprende de la tabla 6.28 que, para utilizaciones bajas, la combinación obtenida consigue tiempos iguales a los obtenidos por el test RTA, no introduciendo ningún tipo de error, siendo el error un

parámetro que marca, en tanto por ciento, la diferencia con la combinación óptima real. Sin embargo, a medida que la utilización del nodo aumenta, el error se incrementa también.

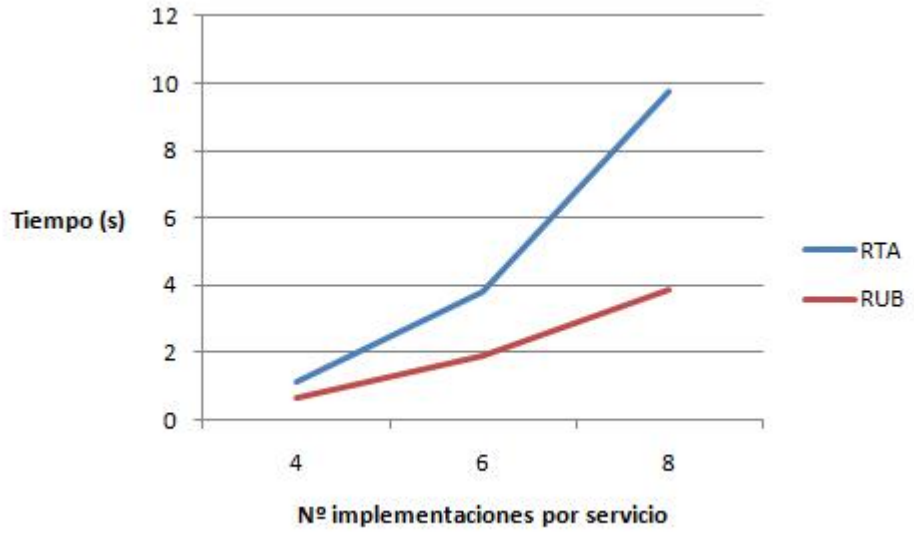

Figura 6.6: Tiempo de ejecución. App. sencilla de tres servicios y utilización previa al 30 %

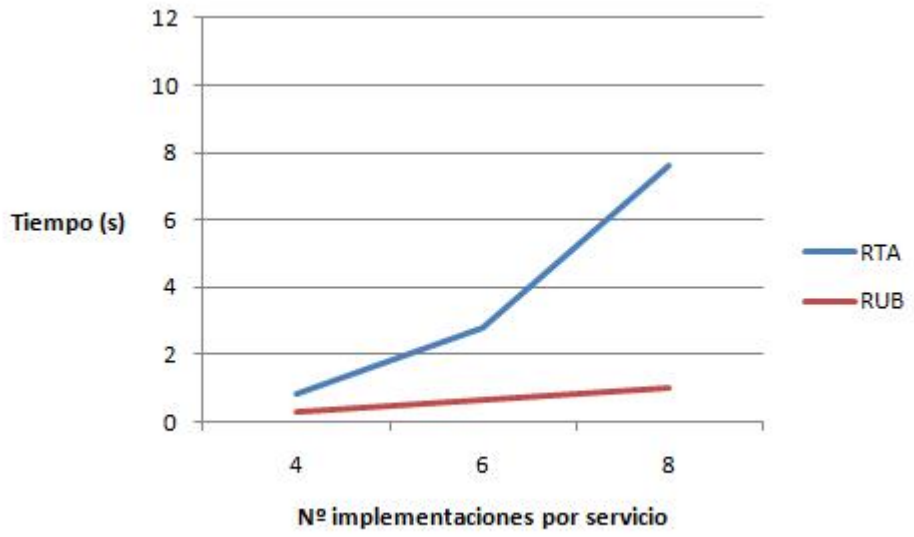

Figura 6.7: Tiempo de ejecución. App. sencilla de tres servicios y utilización previa al 55 %

Cabe destacar los casos en los que no se encuentra combinación alguna debido a que, como se explicó anteriormente, la mayoría de las implementaciones son descartadas por superar la unidad en utilización. Así, el error es nulo, ya que no se puede calcular.

Por último, en lo que se refiere al tiempo de ejecución del programa desarrollado, como se observa de las gráficas 6.5, 6.6 y 6.7, el tiempo para conseguir una solución con el test de planificabilidad RTA son mayores que con el test de planificabilidad RUB. Los tres casos representan tres utilizaciones previas del nodo, y se ve como afecta esto en la obtención de la solución final, incrementando el tiempo. Por otro lado, los casos en los que los nodos no permiten la planificabilidad del sistema, por superar la unidad en la utilización, suponen una reducción del tiempo final de ejecución del programa, pero estando siempre el tiempo del test RUB por debajo del tiempo del test RTA. Además se puede observar como al aumentar las implementaciones por servicio se dispara el tiempo que se tarda en encontrar una solución por el método exhaustivo, por lo que se puede afirmar, de nuevo, que al aumentar la cantidad de implementaciones por servicio, el coste computacional es mayor.

6.2.2.2. Aplicación sencilla. Cinco servicios en serie

Esta segunda ejecución se realiza sobre una aplicación sencilla, compuesta por cinco servicios en serie, "A", "B1", "C", "B2" y "D". El plazo de la aplicación fijado es de 5000ms. Hay que tener en cuenta que el número de combinaciones existentes para cada caso varía dependiendo del número de implementaciones por servicio y del número de servicios usados en la aplicación; para este caso, en el que los resultados se han obtenido a partir de 4, 6 y 8 implementaciones, el número de combinaciones aumenta a 384, 3240 y 14336, respectivamente.

		4 Imp/Servicio (EC's)	6 Imp/Servicio (EC's)	8 Imp/Servicio (EC's)
Nodos al 0 %	RTA	16	16	16
	RUB	16	16	16
	Error (%)	0	0	0
Nodos al 30 %	RTA	22	23 (22,7)	24 (23,1)
	RUB	27 (26,4)	29	29 (28,75)
	Error (%)	22,73	26,09	20,83
Nodos al 55 %	RTA	29	31 (30,43)	32 (31,5)
	RUB	0	0	0
	Error (%)	-	-	-

Tabla 6.29: Datos relativos a los tiempos de respuesta y error introducido de una aplicación sencilla de cinco servicios en serie

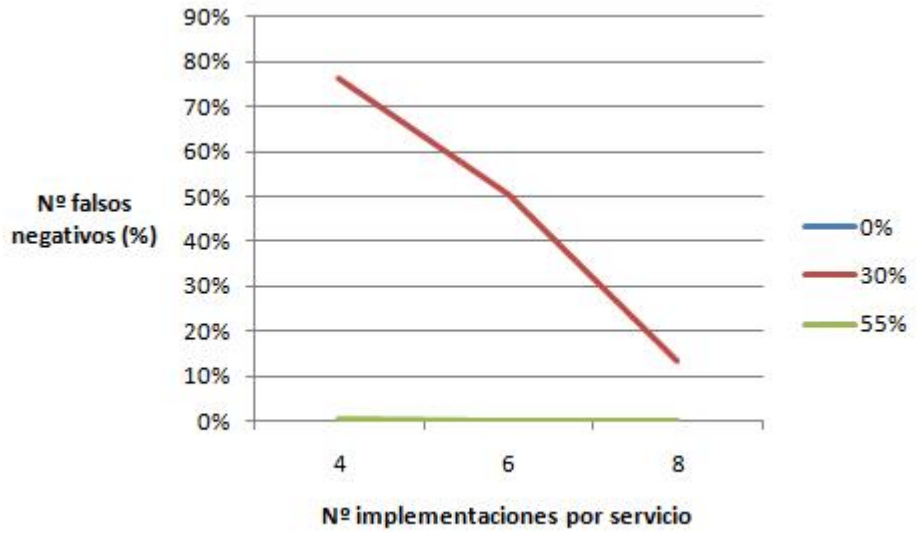

Figura 6.8: Falsos negativos relativos a una aplicación sencilla de cinco servicios

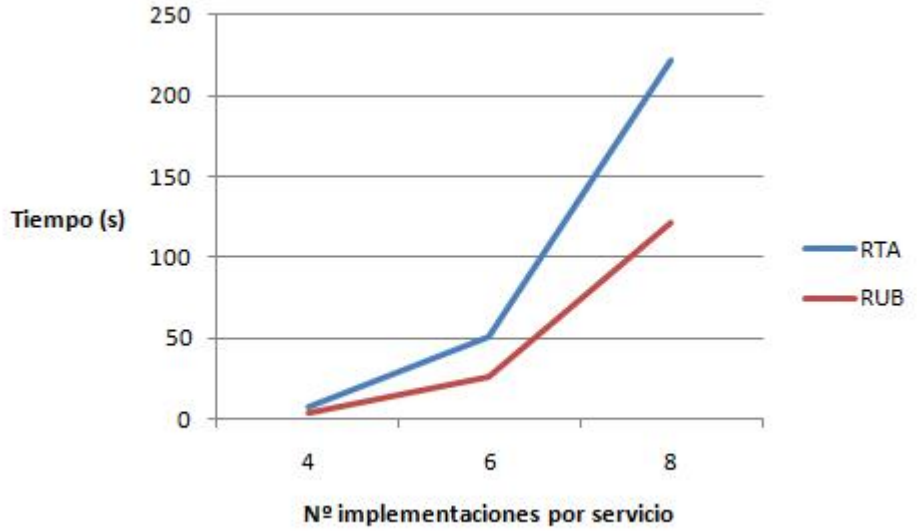

Figura 6.9: Tiempo de ejecución. App. sencilla de cinco servicios y utilización previa al 0 %

Si se analiza la gráfica 6.8, se ve que otro factor que interviene en que el test de planificabilidad RUB es la complejidad de la aplicación, sumado a lo que se comentó en el apartado anterior de la utilización

de los nodos. En este caso, y por estos dos factores unidos, obtenemos datos útiles únicamente en el caso de utilización previa del 30 %.

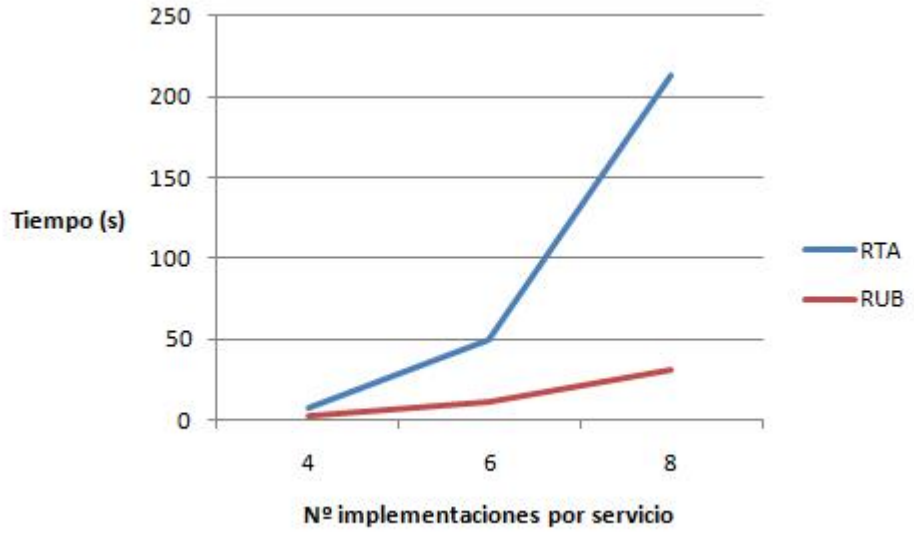

Figura 6.10: Tiempo de ejecución. App. sencilla de cinco servicios y utilización previa al 30 %

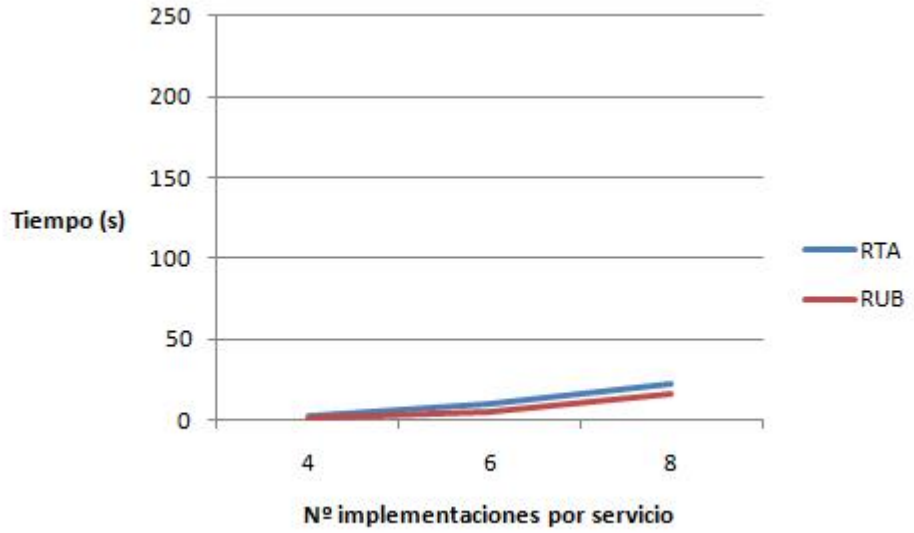

Figura 6.11: Tiempo de ejecución. App. sencilla de cinco servicios y utilización previa al 55 %

Por otro lado, en la tabla 6.29 se observa como afecta el numero de servicios requeridos por el cliente en el error introducido por el test de planificabilidad RUB. Mientras que sigue manteniendo error nulo para utilizaciones bajas, vemos un ligero aumento de éste para utilizaciones medias. Sin embargo, al tener una utilización alta, se ha pasado a no obtener soluciones mediante los test de planificabilidad RUB.

En cuanto al tiempo tardado para la obtención de las soluciones, se ve en las figuras 6.9, 6.10 y 6.11, que las conclusiones que se pueden desprender son las mismas que el caso sencillo de tres servicios en serie, salvo que conlleva tiempos mayores, pero manteniendo la relación entre test RTA y test RUB. De nuevo, como el caso anterior, el tiempo se dispara para mayor número de tareas, aumentando la diferencia entre los dos tests a medida que crece el número de implementaciones por servicio.

6.2.2.3. Aplicación compleja. Cinco servicios con paralelos

Por último, esta ejecución se realiza sobre una aplicación compleja, compuesta por cinco servicios en paralelo, "A1", "B", "D", "A2" y "C". El plazo de la aplicación fijado es de 5000ms. Hay que tener en cuenta que el número de combinaciones existentes para cada caso varía dependiendo del número de implementaciones por servicio y del número de servicios usados en la aplicación; para este caso, en el que los resultados se han obtenido a partir de 4, 6 y 8 implementaciones, el número de combinaciones es de 384, 3240 y 14336, respectivamente.

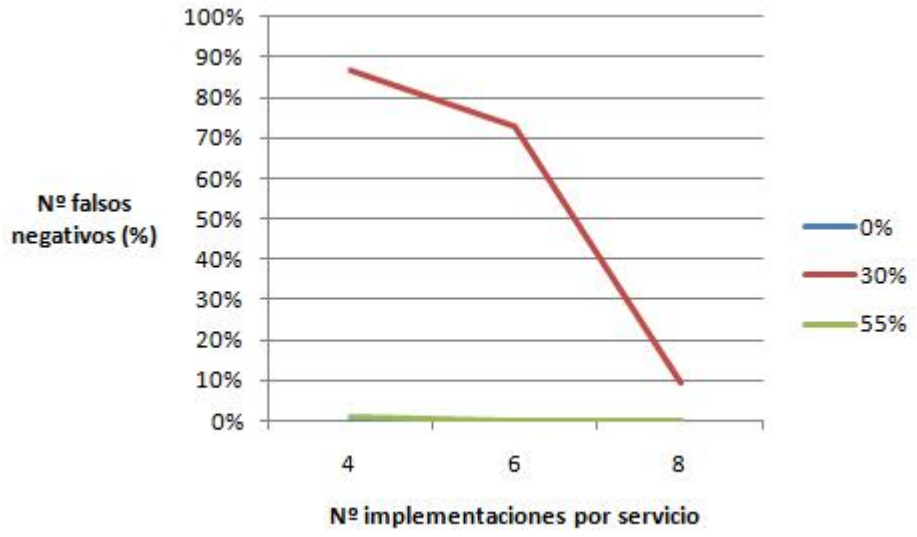

Figura 6.12: Falsos negativos relativos a una aplicación compleja con cinco servicios

Como se ve en la gráfica 6.12, las conclusiones que se pueden sacar son las mismas que el caso anterior, ya que el número de servicios utilizados es el mismo. Así, se puede afirmar que la complejidad de la

aplicación no influye directamente sobre el test de planificabilidad RUB, pero sí el número de servicios que la forman.

		4 Imp/Servicio (EC's)	6 Imp/Servicio (EC's)	8 Imp/Servicio (EC's)
Nodos al 0 %	RTA	13 (12,2)	12	12
	RUB	13 (12,2)	12	12
	Error (%)	0	0	0
Nodos al 30 %	RTA	17	18 (17,7)	17 (16,7)
	RUB	22 (21,5)	22 (21,29)	22
	Error (%)	29,41	22,22	29,41
Nodos al 55 %	RTA	24 (23,38)	26 (25,23)	25 (24,3)
	RUB	0	0	0
	Error (%)	-	-	-

Tabla 6.30: Datos relativos a los tiempos de respuesta y error introducido de una aplicación compleja con cinco servicios

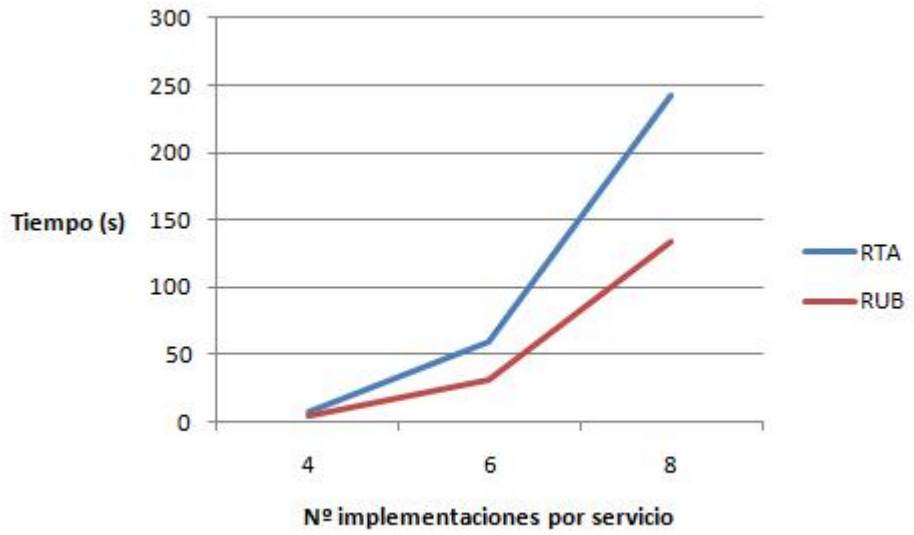

Figura 6.13: Tiempo de ejecución. App. compleja de cinco servicios y utilización previa al 0 %

En la tabla 6.30 se observa como los resultados son semejantes a los obtenidos en una aplicación sencilla de la misma cantidad de servicios. Por tanto, se puede concluir que el número de implementaciones influye directamente en el error introducido por los test RUB, pero no la disposición de los servicios en la aplicación del cliente.

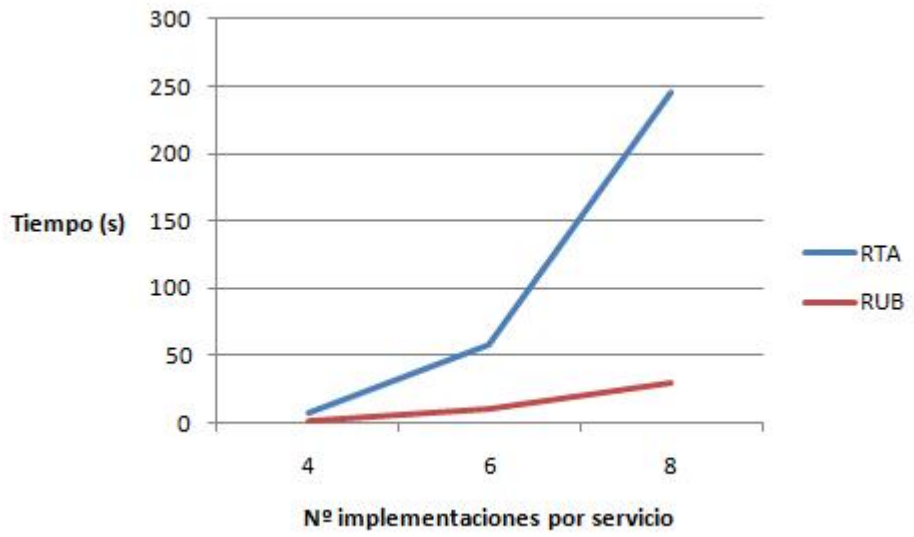

Figura 6.14: Tiempo de ejecución. App. compleja de cinco servicios y utilización previa al 30 %

Finalmente, en referencia al tiempo empleado en la consecución de las soluciones, y que se ve reflejado en las figuras 6.13, 6.14 y 6.15, se puede afirmar que los tiempos aumentan ligeramente, pero se mantienen de valor parecido a los obtenidos en el apartado anterior.

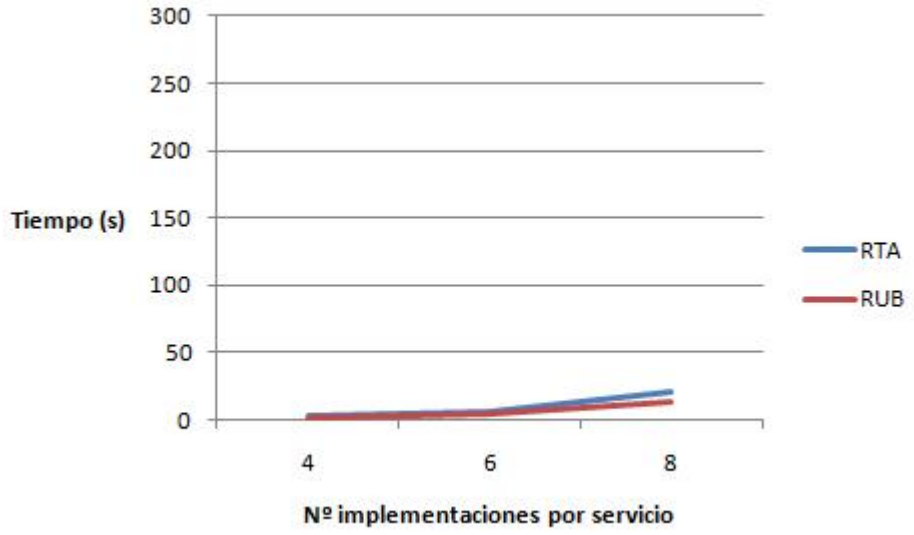

Figura 6.15: Tiempo de ejecución. App. compleja de cinco servicios y utilización previa al 55 %

6.2.2.4. Conclusiones

Las principales conclusiones que se sacan en claro de la prueba 2 son:

- El número de falsos negativos aumenta con el número de implementaciones por servicio, pero existen limitaciones:
  - En primer lugar, si el número de implementaciones se hace lo bastante alto, el sistema empieza a declarar una menor cantidad falsos negativos producida en realidad porque hay menos combinaciones a analizar.
  - La segunda limitación viene dada por la utilización del nodo. En caso de una alta utilización del nodo, también decrece el número de falsos negativos como consecuencia de haber menos implementaciones eliminadas por el control de admisión de utilización.
- El error introducido por el test de planificabilidad RUB es nulo para utilizaciones bajas, mientras que a medida que aumenta la carga previa de los nodos, el error se ve incrementado.
- El tiempo empleado para la consecución de la combinación subóptima siempre queda por debajo del tiempo para obtener la combinación óptima, y esta diferencia es mayor a medida que aumenta el número de implementaciones por servicio. Sin embargo, para utilización alta, y por los mismos motivos que antes, los tiempos son parecidos, al existir realmente menos combinaciones.

En resumen, se puede afirmar que para aplicaciones sencillas las mejoras conseguidas por cambiar el tipo de test de planificabilidad son prácticamente nulas, por tanto, su utilización es irrelevante. Sin embargo, para aplicaciones más complejas, hay que diferenciar entre utilizaciones bajas y altas de los nodos de la red.

El sistema diseñado, utilizando test de planificabilidad RUB mejora notablemente las prestaciones del sistema para aplicaciones que se diseñen en una red de nodos que no tengan un porcentaje de utilización elevado. Es decir, es recomendable la sustitución del test si se tarta de una red de utilización media para conseguir prestaciones similares, a cambio de un coste computacional menor, y de un tiempo dedicado también menor.

6.2.3. Prueba 3

Esta última prueba se aplica de manera previa a la búsqueda exhaustiva de la solución de la aplicación propuesta y posterior a la comprobación individual de tareas por utilización. De esta forma, primero se elimina aquellas implementaciones de servicio que hagan mayor de la unidad la utilización del nodo de manera individual. A continuación, se hace una comprobación individual con tests de planificabilidad RUB para las tareas restantes, eliminando aquellas implementaciones de servicio que no sean planificables por sí solas. Finalmente, con un número reducido de implementaciones, se pasa a realizar la búsqueda exhaustiva.

Con estos cambios, generamos un control de admisión regulado por la utilización previa, que siempre se utiliza, y por los tests de planificabilidad RUB, eliminando muchos casos que luego no serían posibles como solución, reduciendo así el coste computacional del sistema.

Para la ejecución de esta prueba, se aplican las mismas condiciones previas que en las pruebas anteriores, como promediar tras 10 repeticiones.

6.2.3.1. Aplicación sencilla. Tres servicios en serie

Lo primero que se puede desprender de los resultados obtenidos en esta prueba es que cuando los nodos están vacíos o a media utilización, el control de admisión no produce recortes, es decir, todas las implementaciones de servicio pasan satisfactoriamente el test de planificabilidad RUB, teniendo un total de 64, 216 y 512 combinaciones posibles para evaluar de manera exhaustiva, exactamente las mismas que sin emplear el control de admisión. Sin embargo, en el caso de utilización alta en el nodo, el control de admisión comienza a eliminar algunas implementaciones, reduciendo exponencialmente el número de combinaciones finales, y, por consiguiente, reduciendo el tiempo y el coste computacional de obtener la combinación óptima dada una aplicación.

	4 Imp/Servicio	6 Imp/Servicio	8 Imp/Servicio
Tiempo RTA (s)	0,856	2,800	7,614
Tiempo RUB (s)	0,274	0,646	1,008
Tiempo Ctrl. Admisión + RTA (s)	0,342	1,371	3,722
Nº comb. reducido / Nº total	18 / 64	94 / 216	287 / 512

Tabla 6.31: Ctrl. de admisión con utilización de los nodos al 55 % para 3 servicios en serie

En la tabla 6.31 se exponen los resultados para el caso de utilización previa en los nodos del 55 %, para distinto número de implementaciones por servicio. Como se desprende, el número de combinaciones total se ve reducido, obteniendo tiempos mucho menores.

6.2.3.2. Aplicación sencilla. Cinco servicios en serie

De nuevo, y por los mismos motivos que en el caso anterior, el control de admisión no produce recortes, es decir, todas las implementaciones de servicio pasan satisfactoriamente el test de planificabilidad RUB, teniendo un total de 384, 3240, y 14336 combinaciones posibles para evaluar de manera exhaustiva.

En la tabla 6.32 se exponen los resultados para el caso de utilización previa en los nodos del 55 %, para distinto número de implementaciones por servicio, que es el caso que arroja resultados útiles. Como se desprende, el número de combinaciones total se ve reducido, obteniendo tiempos mucho menores.

	4 Imp/Servicio	6 Imp/Servicio	8 Imp/Servicio
Tiempo RTA (s)	2,803	9,422	22,781
Tiempo RUB (s)	1,547	4,554	15,824
Tiempo Ctrl. Admisión + RTA (s)	0,536	5,825	11,15
Nº comb. reducido / Nº total	76 / 384	1658 / 3240	6666 / 14336

Tabla 6.32: Ctrl. de admisión con utilización de los nodos al 55 % para cinco servicios en serie

6.2.3.3. Aplicación compleja. Cinco servicios con paralelos

Como en los casos anteriores, el control de admisión no produce recortes, es decir, todas las implementaciones de servicio pasan satisfactoriamente el test de planificabilidad RUB, teniendo un total de 384, 3240, y 14336 combinaciones posibles para evaluar de manera exhaustiva. Y de nuevo, si que se ve una reducción del coste computacional y del tiempo empleado en nodos de alta utilización.

En la tabla 6.33 se exponen los resultados para el caso de utilización previa en los nodos del 55 %, para distinto numero de implementaciones por servicio. Como se desprende, el número de combinaciones total se ve reducido, obteniendo tiempos mucho menores.

	4 Imp/Servicio	6 Imp/Servicio	8 Imp/Servicio
Tiempo RTA (s)	2,389	6,535	21,089
Tiempo RUB (s)	1,416	3,869	13,276
Tiempo Ctrl. Admisión + RTA (s)	0,507	4,014	13,061
Nº comb. reducido / Nº total	50 / 384	1400 / 3240	6015 / 14336

Tabla 6.33: Ctrl. de admisión con utilización de los nodos al 55 % para cinco servicios con paralelos

6.2.3.4. Conclusiones

A la vista de los resultados, el control de admisión de implementaciones por test de planificabilidad RUB es bastante útil en el sentido de que se ahorra una gran cantidad de tiempo y coste computacional para el sistema para cargas altas de utilización de los nodos de la red.

Sin embargo, cabe destacar que si se empleara el test de planificabilidad RUB después de realizar el control de admisión por test RUB, el tiempo sería aún menor, con la contra de tener la posibilidad de no encontrar la combinación óptima.

En contraste con las pruebas anteriores, en las que se demostraba que para dichas cargas de utilización no era práctico el sistema con test de planificabilidad RUB, ahora sí que lo es. Es decir, en caso de existir una red con alta utilización, se puede aplicar un control de admisión por test RUB para filtrar implementaciones y reducir el tiempo empleado en conseguir una combinación óptima con el método exhaustivo, ya que el método con test de planificabilidad RUB no se puede emplear.

6.2.4. Caso de estudio

Como prueba adicional para la realización de este proyecto, se ha decidido incluir una simulación con datos reales obtenidos a partir del artículo de *Haibo Zeng y Marco Di Natale* [31]. En el documento se proponen tareas de servicios distintos, que nosotros agrupamos en tres: A, B y C. Los servicios están repartidos en tres nodos, denominados ECU, según muestra la figura 6.17, correspondiente al artículo. La figura 6.19 es una versión esquemática de cómo queda la aplicación propuesta, y la figura 6.18 simplificada de cómo queda el sistema completo.

Puesto que cada tarea tiene periodo diferente, se establece el plazo y periodo de todas las tareas en 50000 microsegundos, por lo que se tienen que reajustar los valores de tiempo de ejecución. Además, en el caso del nodo 2, se crea una única tarea que represente a todas las subtareas que se observan, adaptando sus tiempos de ejecución en el peor caso según corresponda. Los ECU, a priori, tiene utilización nula, con lo cual dicho parámetro solo aumenta con el procesamiento de las tareas que interesan.

o_i	$T_i(ms)$	O_i	P_i	$E_i^{\max}(ms)$	o_i	$T_i(ms)$	O_i	P_i	$E_i^{\max}$/size
τ_{59}	100	0	2(2)	2.331	τ_9	10	0	1(16)	$2.331ms$
τ_{63}	50	0	11(16)	13.314	τ_{53}	50	0	1(1)	$0.252ms$
τ_{91}	100	0	13(16)	5.656	m_1	50	0	13(106)	8bytes
τ_{31}	40	0	6(16)	2.163	m_2	50	0	16(30)	7bytes

Figura 6.16: Tabla de datos del artículo de Zeng y Di Natale

A la hora de realizar el experimento, las subtareas del nodo 2 se agrupan en una sola, manera que los datos resultantes son los siguientes:

Servicio	ID Tarea	Nodo	WCET: C (us)	Periodo: T (ms)	Prioridad: p
A	59	1	1165,5	50000	2
B	6391319	2	30500,75	50000	1
C	53	3	252	50000	1

Tabla 6.34: Datos relativos a las tareas resultantes para su uso como datos reales.

Las tareas 63, 91, 31 y 9, alojadas en el nodo 2 y con tiempos se ejecución en el peor caso reflejados en la figura 6.16 (en la casilla $E_i^{max}(ms)$), se unifican en la tarea 6391319 con tiempo de ejecución promediado y de valor 30500,75 microsegundos.

Una vez establecidos los parámetros, se procede como en los casos simulados en apartados anteriores. Se crean distintas implementaciones con las funciones desarrolladas, cumpliendo que los tiempos de ejecución en el peor caso de cada una de las implementaciones generadas están comprendidos entre la mitad y la totalidad de los valores preestablecidos, es decir, los tiempos están entre un 50 % y un 100 % de los

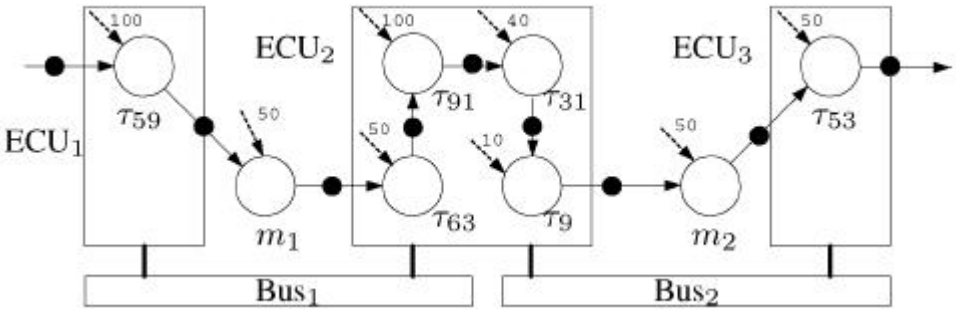

Figura 6.17: Aplicación propuesta en el artículo de Zeng y Di Natale

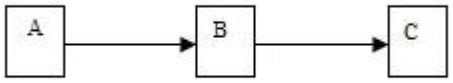

Figura 6.18: Versión simplificada del sistema propuesto por Zeng y Di Natale

valores del artículo y están reflejados en la figura 6.34. Posteriormente, se hace la búsqueda de las mejores implementaciones para la aplicación, las cuales han quedado ordenadas en un árbol generado, consiguiendo distintas combinaciones y analizando cuál es la más adecuada para la aplicación de la figura 6.19.

Para obtener resultados concluyentes, se procede como en las pruebas de los apartados anteriores. Se realiza la ejecución 10 veces, siendo los resultados finales la media de los obtenidos en cada iteración, y partiendo de **4 implementaciones para cada servicio**.

	Tiempo	EC's	Combinación
Caso de estudio	0,086	8	Única: [A59, B6391319, C53]
Caso simulado aleatoriamente	0,989	6 ó 7	Variable

Tabla 6.35: Resultados relativos al experimento propuesto a partir del artículo de Zeng y Di Natale.

De la tabla 6.35 se desprende que, los resultados obtenidos para un sistema que explora un árbol con diversas implementaciones para cada servicio encuentra diferentes combinaciones válidas que implican un menor tiempo de procesamiento en ciclos elementales.

En el caso de estudio propuesto en el artículo adaptado al sistema diseñado, la combinación obtenida es siempre la misma, ya que sólo tenemos una implementación disponible para cada servicio. Sin embargo, al generar tareas con tiempos de ejecución de al menos el 50 % de su homóloga en el artículo, y partir de

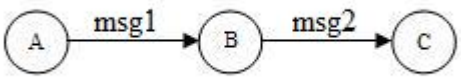

Figura 6.19: Versión esquemática y simplificada de la aplicación propuesta por Zeng y Di Natale

4 implementaciones para cada servicio, la combinación encontrada para la aplicación final es diferente en cada iteración, lo que demuestra el carácter aleatorio del programa desarrollado.

En referente al tiempo que tarda el sistema en converger en una combinación óptima, la simulación llega a tardar del orden de 10 veces más que el caso del artículo, aumento bastante importante tratándose de una aplicación bastante sencilla en cuanto a su composición.

Cabe destacar que al intentar realizar las pruebas aumentando la utilización de los ECU, a pesar de que éstos no superan la unidad en dicho parámetro, los resultados no son útiles porque las tareas existentes no permiten una planificación válida para seguir adelante con la aplicación.

En conclusión, si se aplica el caso propuesto por *Zeng y Di Natale* al sistema desarrollado en este proyecto se obtienen combinaciones que mejoran el tiempo de respuesta, teniendo en cuenta la acotación existente del tiempo de ejecución. Sin embargo, esta mejora sacrifica el tiempo empleado para dicho fin, aumentándolo considerablemente. Por tanto, para este caso de estudio, el uso de los algoritmos desarrollados es recomendable si lo que premia es la reducción del tiempo de respuesta de la aplicación a pesar el tiempo que se tarda en encontrar la combinación óptima.

Capítulo 7

Conclusiones y lineas futuras

Este capítulo resume las ideas que se han planteado a lo largo del proyecto, y se realiza una propuesta de las líneas de posibles trabajos futuros.

7.1. Conclusiones generales

Este trabajo está centrado en aplicar soluciones de flexibilidad de paradigmas ya implementados, a sistemas distribuidos de tiempo real. Así, se han implementado algoritmos a partir de los estudios iniciados por *Estévez Ayres* [9]. Los principales objetivos son asegurar garantías temporales a las aplicaciones y dotarlas de flexibilidad para ofrecer mejores calidades de servicio y tener más tolerancia a fallos. Para ello, se ha pretendido demostrar qué características del paradigma de orientación a servicios se pueden adaptar a sistemas de tiempo real distribuidos.

Para conseguir este fin, se ha seguido el diseño del trabajo de *Iria Estévez Ayres* [9], es decir, se ha adoptado el modelo de sistema, servicio, y aplicaciones utilizado en la tesis. Además, estos diseños, así como los algoritmos adaptados de la tesis, han sido implementados en lenguaje Python. Estos algoritmos de composición de aplicaciones, mediante tests de planificabilidad, permiten seleccionar implementaciones de servicio para la creación de aplicaciones y aseguran las prestaciones del sistema, en cuanto a los nodos físicos y las redes que los unen.

Los estudios realizados en el presente proyecto, así como las principales conclusiones, son:

1. Tras el estudio de los algoritmos usados y diseñados por *Iria Estévez Ayres* en [9], se llega a las conclusión de que un algoritmo heurístico es mucho más eficiente, ya que, con un tiempo y coste computacional mucho menor que con el algoritmo exhaustivo, se llega a las mismas soluciones óptimas o ligeramente inferiores. En el caso de encontrar la misma solución, se habría conseguido con mucho menos tiempo y un menor coste computacional. En el caso de encontrar lo que se denomina una solución subóptima, se habría conseguido una combinación con prestaciones similares, pero de

nuevo, con menor coste computacional y tiempo usado. Esto lleva a afirmar que es aconsejable el uso de un algoritmo heurístico, ya que se consiguen prestaciones casi idénticas a costa de una mejora del tiempo empleado, sea cual sea la aplicación requerida.

2. La implementación de estos algoritmos al lenguaje de programación Python ha sido satisfactoria. Si bien en un principio dicha implementación no fue tan eficiente, a lo largo de la elaboración de este proyecto fin de carrera se ha ido modificando para conseguir una implementación de los algoritmos que funcionase, ya que, como se comentó en otros capítulos, se trata de un diseño recursivo que, a partir de la solución del diseño original, ha permitido desarrollar mejoras. Así, este código puede ser empleado para simulaciones de sistemas de tiempo real, e incluso puede mejorarse en ciertos ámbitos.

3. Para comprobar el comportamiento de los algoritmos utilizados, se han hecho simulaciones configurando distintas situaciones.

4. Para una mejor interacción con la configuración de los parámetros de los algoritmos, se ha desarrollado un cuestionario que se realiza al inicio de cada ejecución del código. Este bloque de preguntas hace una fácil interacción con el programa, ya que no habría que modificar el código en sí. Por otro lado, también se podría diseñar de manera más intuitiva a través de una interfaz gráfica.

5. La introducción en el proyecto del uso los test de planificabilidad RUB como control de admisión ha sido bastante útil, ya que permite al sistema encontrar una solución subóptima con bastante rapidez. Por tanto, se puede afirmar que el uso de test de planificabilidad RUB mejora incluso las prestaciones conseguidas con el algoritmo heurístico empleado.

7.2. Trabajos Futuros

A partir del camino abierto por *Estévez Ayres [9]* en sus tesis en relación al paradigma de la orientación de servicios en sistemas de tiempo real, este proyecto fin de carrera ha hecho un pequeño avance en el estudio.

7.2.1. Futuras lineas de investigación

Mejoras en los algoritmos.

- Creación de nuevos algoritmos de composición de aplicaciones, así como la mejora de los ya existentes, haciendo el código más eficiente o incluso un mejor diseño del algoritmo.

- Implantación de nuevas figuras de mérito que tengan en cuenta la utilización global del sistema o la calidad de servicio de la aplicación, para ser utilizados en campos concretos.

- Desarrollo de nuevos algoritmos de composición que haga uso de los heurísticos de asignación de prioridades ya existentes y que se evalúe el coste de soportar esta característica.
- Realización de sistemas más complejos que el desarrollado en el capitulo 6 para facilitar el uso por usuarios.

Mejoras en el sistema.

- Dotar de flexibilidad al sistema implementado, es decir, facilitar la composición dinámica de aplicaciones con posibilidad de reconfiguración del sistema en tiempo de ejecución y una mayor tolerancia a fallos.
- Dotar al sistema la capacidad de ejecutarse en un modelo orientado a eventos, o un modelo mixto que este gobernado por tiempo y eventos.
- Investigar la posibilidad de que las transmisiones de los mensajes sean en distintos tipos de redes, y su repercusión en los algoritmos de composición.
- Analizar, para distintas redes de tiempo real, la influencia del ancho de banda utilizado, e incluirlo en las figuras de mérito como un dato más a tener en cuenta en la composición de aplicaciones.

Bibliografía

[1] L. Abeni and G. Buttazzo. Resource reservation in dynamic real–time systems. *Real–Time Systems*, 27(2):123–165, 2004.

[2] L. Abeni and Giorgio Buttazzo. Integrating Multimedia applications in hard real-time systems. In *Proc. of the 19th IEEE Real–Time Systems Symposium*, Madrid, Spain, Dec 1998.

[3] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services. Concepts, Architectures and Applications*. Springer, 2004.

[4] N. Audsley. Deadline Monotonic Scheduling. Technical Report YCS_146, Department of Computer Science, University of York, Sept 1991.

[5] N. Audsley. Optimal Priority Assigment and Feasibility of Static Priority Tasks with Arbitrary Start Times. Technical Report YCS_164, Department of Computer Science, University of York, Dec 1991.

[6] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre–emptive scheduling. *Software Engineering Journal*, 8(5):284–292, Sept 1993.

[7] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real–time scheduling: The deadline–monotonic approach. In *Proc. 8th IEEE Workshop on Real–Time Operating Systems and Software*, Mayo 1991.

[8] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Incorporating unbounded algorithms into predictable real–time systems. *Computer Systems Science and Engineering*, 8(3):80–89, 1991.

[9] Iria Estevez Ayres. *Tecnicas de soporte a la flexibilidad funcional en sistemas embarcados distribuidos de tiempo real*. PhD thesis, Departamento de IngenierÃa TelemÃ¡tica. Universidad Carlos III de Madrid, 2007.

[10] S.K. Baruah, R.R. Howell, and L.E. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic real–time tasks on one processor. *Real–Time Systems*, 2:173–179, 1990.

[11] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard real–time sporadic tasks on one processor. In *Proc. 11th IEEE Real–Time Systems Symposium*, pages 182–190, 1990.

[12] Guillem Bernat. *Specification and Analysis of Weakly Hard Real–Time Systems*. PhD thesis, Department de CiÃšncies MatemÃ tiques i InformÃ tica. Universitat de les Illes Balears, 1998.

[13] R. Bettati and J. Liu. End-to-End Scheduling to Meet Deadlines in Distributed Systems. In *Proc. of the 12th International Conference on Distributed Systems*, pages 452–459, Japan, Jun 1992.

[14] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30:129–154, 2005. 10.1007/s11241-005-0507-9.

[15] Alan Burns, Neil Hayes, and M.F. Richardson. Generating feasible cyclic schedules. *Control Engineering and Practice*, 3(2):151–162, 1995.

[16] Alan Burns and Andy Wellings. *Real–Time Systems and their programming languages*. Addison Wesley, second edition, 1996.

[17] Giorgio Buttazzo. Research trends in real-time computing for embedded systems. *ACM SIGBED Rev.*, 3(3):1–10, July 2006.

[18] Giorgio Buttazzo and L. Abeni. Adaptive workload management through elastic scheduling. *Real Time–Systems*, 23(1–2):7–24, 2002.

[19] Giorgio Buttazzo and L. Abeni. Smooth rate adaptation through impedance control. In *Proc. of the 14th Euromicro Conference on Real–Time Systems*, pages 3–10, Vienna, Austria, 2002.

[20] Giorgio Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. In *IEEE Transactions on Computers*, volume 51 of *3*, pages 289–302, Mar 2002.

[21] Mario Caccamo, Giorgio Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proc. of the 21th IEEE Real–Time Systems Symposium*, pages 295–304, Orlando, FL, USA, Dec 2000.

[22] Mario Calha. *A Holistic Approach Towards Flexible Distributed Systems*. PhD thesis, DET / IEETA-LSE. Universidade de Aveiro, 2006.

[23] Celeste Campo. *TecnologÃas Middleware para el Desarrollo de Servicios en Entornos de Computacion Ubicua*. PhD thesis, Departamento de IngenierÃa TelemÃ¡tica. Universidad Carlos III de Madrid, 2004.

[24] G. Coulouris, J. Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. International Computer Science Series. Addison-Wesley Longman, Inc., 4nd edition edition, Jun 2005.

[25] Robert Davis and Andy J. Wellings. Dual Priority Scheduling. In *IEEE Real-Time Systems Symposium*, pages 100–109, 1995.

[26] Robert I. Davis, Attila Zabos, and Alan Burns. Efficient exact schedulability tests for fixed priority real-time systems. 57:1261–1276, September 2008.

[27] M. L. Dertouzos. Control robotics: the procedural control of physical processses. *Information Processing*, page 74, 1974.

[28] R. Garey and D. Johnson. Complexity Bounds for Multiprocessor Scheduling with Resource Constraints. *SIAM Journal of Computing*, 4(3):187–200, 1975.

[29] R. Garey and S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal of Computing*, 4(4):397–411, 1975.

[30] J.J. Gutierrez-Garcia and Michael Gonzalez-Harbour. Optimized Priority Assignment for Tasks and Messages in Distributed Real–Time Systems. In *Proc. of the 3rd Workshop on Parallel and Distributed Real–Time Systems*, pages 124–132, April 1995.

[31] Marco Di Natale Haibo Zeng. Stochastic analysis of can-based real-time automotive systems. Nov. 2009.

[32] P. K. Harter. Response times in level structured systems. Technical Report CU–CS–269–94, Department of computer Science. University of Colorado, USA, 1984.

[33] P. K. Harter. Response times in level structured systems. *ACM Transactions on Computer Systems*, 5:232–248, 1997.

[34] M. H. Huhns and M. P. Singh. Service-oriented computing: Key concepts and principles. 9(1):75–81, Jan./Feb. 2005.

[35] Steve Jones. Toward an acceptable definition of service. *IEEE Software*, 22(3):87–93, 2005.

[36] M. Joseph and P. Pandya. Finding Response Times in a Real–Time System. *BCS Computer Journal*, 29(5):390–395, Oct 1986.

[37] Hermann Kopetz. *Real–Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, 1997.

[38] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proc. of the 11th IEEE Real-Time Systems Symposium*, pages 201–209, 1990.

[39] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm:exact characterization and average case behaviour. In *Proc. of the 10th IEEE Real-Time Systems Symposium*, pages 166–171, Santa Monica, CA, USA, 1989.

[40] J. Leung and J. Withehead. On the complexity of fixed priority scheduling of periodic real–time tasks. *Performance Evaluation*, 2(4), 1982.

[41] G. Lipari and S.K. Baruah. Greedy reclamation of unused bandwidth in constant bandwidth servers. In *Proc. of the 12th Euromicro Conference on Real–Time Systems*, pages 192–200, Stokholm, Sweden, 2000.

[42] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard–real–time environment. *J. ACM*, 20(1):46–61, 1973.

[43] C. Locke. Software architecture for hard real–time applications: Cyclic executives versus fixed priority executives. *Real–Time Systems*, 4(1):37–53, 1992.

[44] Jose Carlos Palencia. *AnÃ¡lisis de planificabilidad de sistemas distribuidos de tiempo real.* PhD thesis, Grupo de Computadores y Tiempo Real. Universidad de Cantabria, 1999.

[45] Jose Carlos Palencia-Gutierrez and Michael Gonzalez-Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *Real–Time Systems Symposium, 1998.*, pages 26–37, December 1998.

[46] Jose Carlos Palencia-Gutierrez and Michael Gonzalez-Harbour. Exploiting Precedence Relations in the Schedulability Analysis of Distributed Real–Time Systems. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, pages 328–339, December 1999.

[47] Paulo Pedreiras. *Supporting Flexible Real-Time Communication on Distributed Systems.* PhD thesis, Departamento de Electrónica e Telecomunicações of the University of Aveiro, Aveiro, Portugal, 2003.

[48] I. Ripoll. *Planificacion con Prioridades Dinamicas en Sistemas de Tiempo Real CrÃtico.* PhD thesis, Valencia, 1996.

[49] M. Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8(4):10–17, 2001.

[50] Lui Sha, Tarek Abdelzaher, Karl-Erik Arzen, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysious K. Mok. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems Journal*, 28(2/3):101–155, 2004.

[51] Kang G. Shin and P. Ramanathan. Real–time computing: A new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1), 1994.

[52] Marco Spuri and Giorgio C. Buttazzo. Efficient aperiodic service under the earliest deadline scheduling. In *Proc. of the IEEE Real–Time Systems Symposium*, 1994.

[53] Marco Spuri and Giorgio C. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *Real–Time Systems*, 10(2):179–210, 1996.

[54] John A. Stankovic. Misconceptions about real–time computing: A serious problem for next generation systems. *IEEE Computer*, 21(10):10–19, 1988.

[55] John A. Stankovic et al. Implications of Classical Scheduling Results for Real–Time Systems. *IEEE Computer*, 28(6), 1995.

[56] J. Sun and J. Liu. Synchronization protocols in distributed real-time systems. In *Proc. of the 16th International Conference on Distributed Systems*, May 1996.

[57] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 50(2–3):117 – 134, 1994.

[58] Ken Tindell. Adding Time–Offsets to Schedulability Analysis. Technical Report YCS221, Department of Computer Science, University of York, England, 1994.

[59] Ken Tindell, Alan Burns, and Andy Wellings. Allocating hard real–time tasks (an NP–hard problem made easy). *Real–Time Systems*, 4(2):145–165, 1992.

[60] Ken Tindell, Alan Burns, and Andy Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real–Time Systems Journal*, 6(2):133–151, March 1994.

[61] Keneth William Tindell. *Fixed Priority Scheduling of Hard Real–Time Systems.* PhD thesis, University of York, 1993.

[62] World Wide Web Consortium. *Web Services Glossary*, Feb. 2004. W3C Working Group Note.

[63] J. Zamorano-Flores. *Planificacion estatica de procesos en sistemas de tiempo real crÃticos.* PhD thesis, Universidad Politecnica de Madrid, 1995.